

C++ 语言的 设计和演化

The Design
and Evolution
of C++

(美) Bjarne Stroustrup 著

裘宗燕 译



机械工业出版社
China Machine Press



Addison-Wesley

译者序

这是一本独特的书，是由C++语言的设计师本人写的，描述C++语言的发展历史、设计理念及技术细节的著作。在计算机发展的历史上，这种从多方面多角度描述一种主流语言各个方面的综合性著作，至今我还只看到这一本。阅读本书，不仅可以了解有关C++语言的许多重要技术问题和细节，还可以进一步理解各种C++特征的设计出发点、设计过程中所考虑的方方面面问题，以及语言成形过程中的各种权衡与选择。每个学习和使用C++语言的人，一定能由此加深对自己所用工具的认识，进一步理解应该如何用好这个语言，此外还能看到作者对于复杂的系统程序设计的许多观点和想法。如果一个人想深入了解C++语言，想使C++成为自己得心应手的工具，想在复杂的系统程序设计领域中做出一些有价值的工作，想了解面向对象程序设计语言的一般性问题，想了解程序设计语言的发展现状、问题和前景，本书都是最值得阅读的书籍之一。

C++语言的设计目标是提供一种新的系统开发工具，希望能在一些方面比当时的各种工具语言有实质性的进步。今天来看，C++最重要的作用就是使那时的阳春白雪(数据抽象、面向对象的理论和技术等)变成了普通的系统开发人员可以触及、可以接受使用、可以从中获益的东西。这件事在计算科学技术发展的历史记录上必定会留下明显的痕迹。本书从一个最直接参予者的角度，记述了C++语言的起源和发展，记录了它怎样成长为今天的这个语言，怎样使语言研究的成果变成了程序员手中的现实武器。

从来都没有一种完美的程序设计语言。C++语言由于其出身(出自C语言)，由于其发展过程中各种历史和现实因素的影响，也带着许多瑕疵和不和谐，尤其是在作为C++基础的C语言的低级成分与面向数据抽象的高级机制之间。对于一个目标是支持范围广泛的复杂系统实现的语言来说，这类问题也很难避免。为了系统的效率和资源的有效利用，人们希望有更直接的控制手段(低级机制)；而为了将复杂的功能组织成人能够理解和把握的系统，又需要有高级的机制和结构。在使用一个同时提供了这两方面机制的语言时，应该如何合理而有效地利用它们，使之能互为补充而不是互相冲突，本书中许多地方讨论到这些问题，也提出了许多建议。这些，对于正确合理地使用C++语言都是极其重要的。

C++并不是每个人都喜欢的语言(没有任何语言可能做到这一点)，但不抱偏见的人都会承认，C++语言取得了极大的成功。C++语言的工作开始于一个人(本书作者)的某种很合理、很直观的简单想法(为复杂的系统程序设计提供一种更好的工具)，由于一个人始终不渝的努力，一小批人的积极参予，在一大批人(遍及世界的系统开发人员)的热心关注、评论和监督下，最终造就出了一项重要的工作。这个工作过程本身就非常耐人寻味，它也是在现代信息环境(主要是因特网)下，开展全球范围的科学技术研究的一个最早的演练。在这个成功中，商业的考虑、宣传和炒作从来没有起过任何实质性的作用，起作用的仍然是理性的思维、严肃的科学态度、无休止的踏踏实实的实际工作。这些，与今天在信息科学技术领域中常见的浮躁情绪和过分的利益追求形成了鲜明对比。许多事实给了我们一种警示：时尚转眼就可能变成无人理睬的烂泥，仅仅被眼球注意的东西很快就会被忽视，炒作最凶的东西往往也消失得最快，

而真正有价值的成果则往往起源于人们最基本的需求和向往。

作为C++语言的创造者，作者对于自己的作品自然是珍爱有加。对某些针对C++语言的批评，本书中也有一些针锋相对的比较尖锐的观点。但通观全书，作者的论点和意见还是比较客观的，并没有什么过于情绪化的东西。在前瞻性讨论中，作者提出了许多预见。经过这五六年时间，其中一些已经变成了现实，也有些，例如特别有效的开发环境，还在发展之中。这些可能也说明了语言本身的一些性质：C++是个比较复杂的语言，做好支持它的工具绝不是一件容易的事情，在这些方面还有许多发展余地。

作者在讨论C++的设计和发展的过程中，还提出了许多人文科学领域的问题，提出了他在从事科学技术工作中的人文思考，其认识和观点也是C++成长为今天这样一个语言的基础。作者的这些想法也可以供我们参考。

今天，作为一种通用的系统程序设计语言，C++已经得到了广泛的认可。许多个人和企业将C++作为软件系统的开发工具，许多计算机专业课程用它作为工具语言。近十年来，国外的一些计算机教育工作者也一直在探索将C++作为CSI(计算机科学的第一门课程)的工作语言的可行性，国内学习和使用C++的人也越来越多。在这种情况下，由C++语言设计师Bjarne Stroustrup本人撰写的有关C++的两部重要著作，本书和《C++程序设计语言》，即将在中国出版，这当然是非常有意义的事情。为此我非常感谢机械工业出版社的管理和编辑人员(相信许多计算机工作者也会如此)，感谢他们在国内出版界更多关注时髦计算机图书的浪潮中，愿意付诸努力，出版一些深刻的、影响长远的重要著作。我祝愿这种工作能获得丰厚的回报，对于整个社会，也包括出版社自身。

作为译者，我希望作为自己工作结果的这个中译本能给学习C++语言、用这个语言从事教学、从事程序设计工作和复杂系统程序设计的人们提供一点帮助，使这本有关C++语言的最重要著作中阐述的事实和思想能够被更多人所了解。虽然我始终将这些铭记在心，但译文中仍难免出现差错和疏漏，在此也恳请有见识的读者不吝赐教。

裘宗燕

2001年6月于北大



裘宗燕 北京大学数学学院信息科学系教授。关心的主要学术领域包括计算机软件理论、程序设计方法学、程序设计语言和符号计算。已出版多部著作和译著，包括《程序设计语言基础》(译著，1990)，《Mathematica数学软件系统的应用与程序设计》(1994)，《从问题到程序——程序设计与C语言引论》(1999)，《程序设计实践》(译著，2000)等。

e-mail: qzy@math.pku.edu.cn

前　　言

一个人，如果不耕作，
就必须写作。

——Martin A. Hansen

ACM关于程序设计语言历史的HOLP-2会议要我写一篇关于C++历史的文章。这看起来是一个很合理的想法，还带着点荣誉性质，于是我就开始写了。为了对C++的成长有一个更全面更公平的观点，我向一些朋友咨询了他们对C++那些早期日子的记忆。这就使关于这项工作的小道消息不胫而走。有关的故事逐渐变了味，有一天，我忽然接到一个朋友的来函，问我在哪里可以买到我关于C++设计的新书。这个电子邮件就是本书的真正起源。

在传统上，关于程序设计和程序设计语言的书都是在解释某种语言究竟是什么，还有就是如何去使用它。但无论如何，有许多人也很想知道某个语言为什么会有它现在这个样子，以及它是怎样成为这个样子的。本书就是针对C++语言，想给出对后面这两个问题的解释。在这里要解释C++怎样从它的初始设计演化到今天的这个语言，要描述造就了C++的各种关键性的问题、设计目标、语言思想和各种约束条件，以及这些东西又是如何随着时间的推移而变化的。

自然，C++语言和造就它的设计思想、编程思想本身并不会演化，真正演化的是C++用户们对于实际问题的理解，以及他们对于为了帮助解决这些问题而需要的工具的理解。因此，在本书中也将追溯人们用C++去处理的各种关键性问题，以及实际处理那些问题的人们的认识，这些都对C++产生了重要影响。

C++仍然是一个年轻的语言，许多用户对这里将要讨论的一些问题还知晓。这里所描述的各种决策的进一步推论，可能还需要一些年才能变得更清晰起来。本书要展示的是我个人关于C++如何出现、它是什么以及它应该是什么的观点。我希望这些东西能够帮助人们理解怎样才能最好地使用C++，理解C++的正在继续进行的演化进程。

书中特别要强调的是整体的设计目标、实际的约束以及造就出C++的那些人们。有关各种语言特征的关键性设计决策的讨论被放到了相应的历史环境里。在这里追溯了C++的演化过程，从带类的C开始，经过Release 1.0和2.0，直到当前ANSI/ISO的标准化工作，讨论了使用、关注、商业行为、编译系统、工具、环境和库的爆炸性增长，还讨论了C++与C和Simula关系的许多细节。对于C++与其他语言的关系只做了简短讨论。对主要语言功能的设计，例如类、继承、抽象类、重载、存储管理、模板、异常处理、运行时类型信息和名字空间等，都在一定细节程度上进行了讨论。

本书的根本目的，就是想帮助C++程序员更好地认识他们的语言、该语言的背景和基本概念；希望能激励他们去试验那些对他们来说还是新的C++使用方式。本书也可供有经验的程序员和程序设计语言的学生阅读，可能帮助他们确定使用C++是不是一件值得做的事情。

致谢

我非常感谢Steve Clamage, Tony Hansen, Lorraine Juhl, Peter Juhl, Brian Kernighan, Lee Knight, Doug Lea, Doug McIlroy, Barbara Moo, Jens Palsberg, Steve Rumsby和Christopher Skelly。感谢他们完整地阅读了本书的手稿，他们建设性的指教使本书的内容和组织都发生了重要变化。Steve Buroff, Martin Carroll, Sean Corfield, Tom Hagelskjoer, Rick Hollinbeck, Dennis Mancl和Stan Lippmann通过对一些章节的评论提供了帮助。还要感谢Archie Lachner在我还没有想到这本书之前就提出了对本书的要求。

自然，我还应该感谢那些帮助创造出C++语言的人们。从某种意义上说，本书就是献给他们的礼物，他们中一部分人的名字可以在各个章节和索引中找到。如果要我点出一些个人来，那就必然是Brian Kernighan, Andrew Koenig, Doug McIlroy和Jonathan Shopiro。他们中的每一位在过去十多年期间一直支持和鼓励我，也是提供各种想法的源泉。还有，感谢Kristen Nygaard和Dennis Ritchie作为Simula和C的设计师，C++从它们那里借用了一些关键性的成分。经过这些年，我已经逐渐了解到他们不仅是才华横溢的讲究实际的语言设计师，而且也是真正的绅士和绝对亲切的个人。

Bjarne Stroustrup

Murray Hill, New York

致 读 者

写作是绝无仅有的一种
只有通过写才能学到的艺术。
——佚名

本书的主题——怎样读这本书——C++的一个时间表——C++与其他程序设计语言

引言

C++语言的设计就是想为系统程序设计提供Simula的程序组织功能，同时又提供C语言的效率和灵活性。当时是希望在有了这些想法的半年之内就能将它提供给实际项目使用。它成功了。

在那个时候，1979年中期，这个目标的朴实性或者是荒谬性都还没有被认识清楚。说这个目标是朴实的，因为它并不涉及任何创新。说它是荒谬的，无论是从时间的长短还是从对效率和灵活性的苛求。在这些年里也确实出现了一些创新，效率和灵活性得到了维持，没做什么妥协。其间，随着时间推移，C++的目标也进行了精化，经过精炼和推敲，被弄得更加清晰了。今天在使用中的C++正是直接地反映了它的初始目标。

本书的宗旨就是想把这些目标见诸于文字，追溯其演化过程，描述C++是如何从许多人为建立一个语言而做的努力中浮现出来，并按照这些目标为它的用户服务的。为能做到这一点，我将试着在历史事实（例如名字、地点和事件）与语言设计、实现和使用的技术事项之间寻找一种平衡。列出每个小事件并不是我的目的，但也需要关注一些对C++的定义实际产生了影响，或者可能影响其未来发展和使用的重要事件、思想和趋势。

在描述这些事件的时候，我将试着按照当时发生的情况去描述它们，而不是按我或者其他可能更喜欢它们发生的样子。只要合理，我都使用选自文献的引文来说明有关的目标、原理和特征，就像在它们出现的时候那样。我也试着不对事件表现出某种事后的聪明；反之，我总把回顾性的注解和有关一个决策所蕴涵的东西的注解单独写出来，并明确注明这些是回顾。简单说，我非常厌恶修正主义的历史学，想尽量地避免它。例如，当我提到“我那时就发现Pascal的类型系统比没有还要坏——它是一种枷衣，产生的问题比它解决得更多。它迫使我去扭曲自己的设计，以适应一个面向实现的人造物品。”这也就是说，我认为在那个时候这是事实，而且是一个对C++的演化有着重要影响的事实。这种对Pascal的苛刻评价是否公平，或者今天（在十几年之后）我是否还会做出同样的评价与此并无干系。我如果删掉这个事实（比如说，为了不伤害Pascal迷们的感情，或为免除自己的羞愧，或为避免争论）或者修改它（提供一个更完全和调整后的观点），那就是包装了C++的历史。

我试着提及对C++的设计和演化做出了贡献的人们，也试着特别提出他们的贡献以及事情发生的时间。这样做在某种意义上说是很冒险的。因为我并没有完美的记忆，很可能会忽略了某些贡献。我在此表示歉意。我是要提出导致了有关C++的某个决策的人的名字。不可避免，在这里提出的有可能并不都是第一个遇到某个特定问题的人，或第一个想出某种解决方

案的人。这当然很不幸，但含含糊糊或者干脆避免提起人名将更糟糕。请毫不犹豫地给我提供信息，这样做可能有助于澄清某些疑点。

在描述历史事件时总存在着一个问题：我的描述是否客观。我已经试着去矫正自己不可避免的倾向性，去设法获得我没有参与的各种事件的信息，与涉足有关事件的人交谈，并请一些参与了C++演化过程的人们读这本书。他们的名字可以在前言的最后找到。此外，在《程序设计语言的历史》(HOPL-2, History of Programming Languages) 会议论文 [Stroustrup, 1993] 中包含了取自这本书的核心历史事件，它经过广泛审阅，被认为并不包含不适当的倾向性。

怎样读这本书

本书第一部分大致是按照时间顺序审视C++的设计、演化、使用和标准化过程。我选择这种组织方式是因为在前面的一些年里，主要的设计决策可以作为一个整齐的有逻辑性的序列，映射到一个时间表里。第1、2、3章描述了C++的起源以及它从带类的C到Release 1.0的演化。第4章描述了在这期间以及后来指导C++成长的一些原则。第5章提供了一个1.0之后的历史年表。第6章描述了ANSI/ISO标准化的努力。第7、8章讨论了应用、工具和库。最后，第9章给出的是一个回顾和一些面向未来的思考。

第二部分描述的是Release 1.0之后C++的发展。这个语言成长起来了，但还是在Release 1.0前后建造起来的框架之内。这个框架包括了一组所需要的特征，如模板和异常处理，还有指导着它们的设计的一组规则。在Release 1.0之后，年代排列对于C++的发展就不那么重要了，即使在1.0之后的扩充按照年代排列的情况与实际有所不同，C++的定义在实质上也还会是目前这个样子。因此，解决各种问题、提供各种特征的实际顺序就只有历史研究的价值了。严格按照时间顺序进行描述会干扰思想的逻辑流程，所以第二部分是围绕着重要语言特征组织起来的。第二部分的各章都是独立的，因此可以按任意顺序阅读；第10章，存储管理；第11章，重载；第12章，多重继承；第13章，类概念的精练；第14章，强制转换；第15章，模板；第16章，异常处理；第17章，名字空间；第18章，C预处理器。

不同的人对于一本有关程序设计语言的设计和演化的书所抱的期望是大相径庭的。特别地，对于到底应该以怎样的细节程度讨论这个题目，很可能任意两个人都不会有相同的意见。我所收到的有关我的HOPL-2论文不同版本的每份评审意见（大大超过10份）的形式都是“这篇文章太长……请在论题X、Y和Z方面增加一些信息”。更糟的是，大约有三分之一的意见里有这样的见解：“请删掉那些哲学/信仰的废话，给我们提供真正的技术细节”。另外三分之一的见解则是：“让那些无趣的细节绕了我吧，请增加有关你的设计哲学方面的信息”。

为了摆脱这种两难局面，我实际上在一本书里写了另一本书。如果你对各种细节不感兴趣，那么就请首先跳过所有的小节（以x.y.z形式编号的节，其中x是章的编号而y是节的编号），而后再去读那些看起来有兴趣的节。你也可以按顺序读这本书，从第一页开始一直读到结尾。在这样做的时候，你就有可能陷进去，被某些细节缠住。这样说并不意味着细节就不重要。正相反，如果只是考虑原则和一般性，那就根本不可能理解一个程序设计语言。具体实例是最基本的东西。但无论如何，在查看细节时，如果没有能够将它们匹配其中的整体画面，人也很容易深深地陷入迷途。

作为进一步的辅助，在第二部分里，我将主要讨论集中在新特征和公认的高级特征方面，这也就使第一部分能够集中在基础方面。几乎所有关于C++演化的非技术性信息都可以在第一部分里找到。对于“哲学讨论”缺乏耐心的人可以跳过第4章到第9章，转过去看第二部分里有关语言特征的技术细节。

我设想某些人会将本书作为参考文献使用，许多人可能只读一些独立的章而不看前面那些章。为使这种使用也能行得通，我已经把许多章做成对有经验的C++程序员而言是自足的，并通过交叉引用和索引项目使人能更加自由。

请注意，我并没有试图在这里定义C++的各种特征，而只是陈述了足够多的细节，提供了关于这些特征缘何而来的自足的描述。我也不想在这里教C++编程或者设计，如果要找一本教科书，请看 [2nd]。

C++ 时间表

这里的C++时间表可能帮助你看清这个故事将把你带到哪个地方：

1979	3月	开始带类的C (C with Class) 的工作
	10月	第一个带类的C实现投入使用
1980	4月	第一篇关于带类的C的贝尔实验室内部报告[Stroustrup, 1980]
1982	1月	第一篇关于带类的C的外部论文[Stroustrup, 1982]
1983	8月	第一个C++实现投入使用
	12月	C++命名
1984	1月	第一本C++手册
1985	2月	第一次C++外部发布 (Release E)
	10月	Cfront Release 1.0 (第一个商业发布)
	10月	<i>The C++ Programming Language</i> [Stroustrup, 1986]
1986	8月	有关“什么是”的文章 [Stroustrup, 1986]
	9月	第一次OOPSLA会议 (OO的宣传开始时集中在Smalltalk)
	11月	第一个C++的商业移植 (Cfront 1.1, Glockenspiel)
1987	2月	Cfront Release 1.2
	11月	第一次USENIX C++会议 (圣菲, 新墨西哥州)
	12月	第一个GNU C++发布 (1.13)
1988	1月	第一个Oregon Software C++发布
	6月	第一个Zortech C++发布
	10月	第一次USENIX C++实现者工作会议 (Estes Park, 科罗拉多州)
1989	6月	Cfront Release 2.0
	12月	ANSI X3J16组织会议 (华盛顿特区)
1990	5月	第一个Borland C++发布
	3月	第一次ANSI X3J16技术会议 (Somerset, 新泽西州)
	5月	<i>The Annotated C++ Reference Manual</i> [ARM]
	7月	模板被接受 (西雅图, 华盛顿州)

(续)

	11月	异常被接受 (Polo Alto, 加利福尼亚州)
1991	6月	<i>The C++ Programming Language</i> (第2版) [2nd]
	6月	第一次ISO WG21会议 (Lund, 瑞典)
	10月	Cfront Release 3.0 (包括模板)
1992	2月	第一个DEC C++发布 (包括模板和异常)
	3月	第一个Microsoft C++发布
	5月	第一个IBM C++发布 (包括模板和异常)
1993	3月	运行时类型识别被接受 (Portland, 俄勒冈州)
	7月	名字空间被接受 (慕尼黑, 德国)
1994	8月	ANSI/ISO委员会草案登记
1997	7月	<i>The C++ Programming Language</i> (第3版)
	10月	ISO标准通过表决被接受
1998	11月	ISO标准被批准

关注使用和用户们

本书是为C++用户而写的, 也就是说, 为那些程序员和设计师。我已经试图 (无论你相信与否) 在给出一种有关C++语言、它的功能和它的演化过程的用户观点时, 尽量避免那些真正晦涩的深奥论题。有关语言的纯粹技术性讨论, 只有在它们确实阐明了某些对用户有重要影响的问题时, 才在这里展开。有关模板中的名字检索 (15.10节) 和临时量生存期的讨论就是这方面例子。

程序设计语言专门家们、语言律师们以及实现者们将在本书中发现许多珍闻, 但本书的目标更多的是想展现出一幅大范围的图景, 而不是精确详尽的点点细节。如果你希望的是精确的技术细节的话, C++的定义可以从*The Annotated C++ Reference Manual* [ARM]、*The C++ Programming Language* (第2版) [2nd] 以及ANSI/ISO标准化委员会的工作文件中找到。当然, 如果没有对于语言用途的一种认识, 一个语言定义的细节是根本无从详尽理解的。这个语言 (无论其细节还是全部) 的存在就是想有助于程序的构造。我写这本书的意图也就是提供一种洞察力, 能够对这方面的努力有所帮助。

程序设计语言

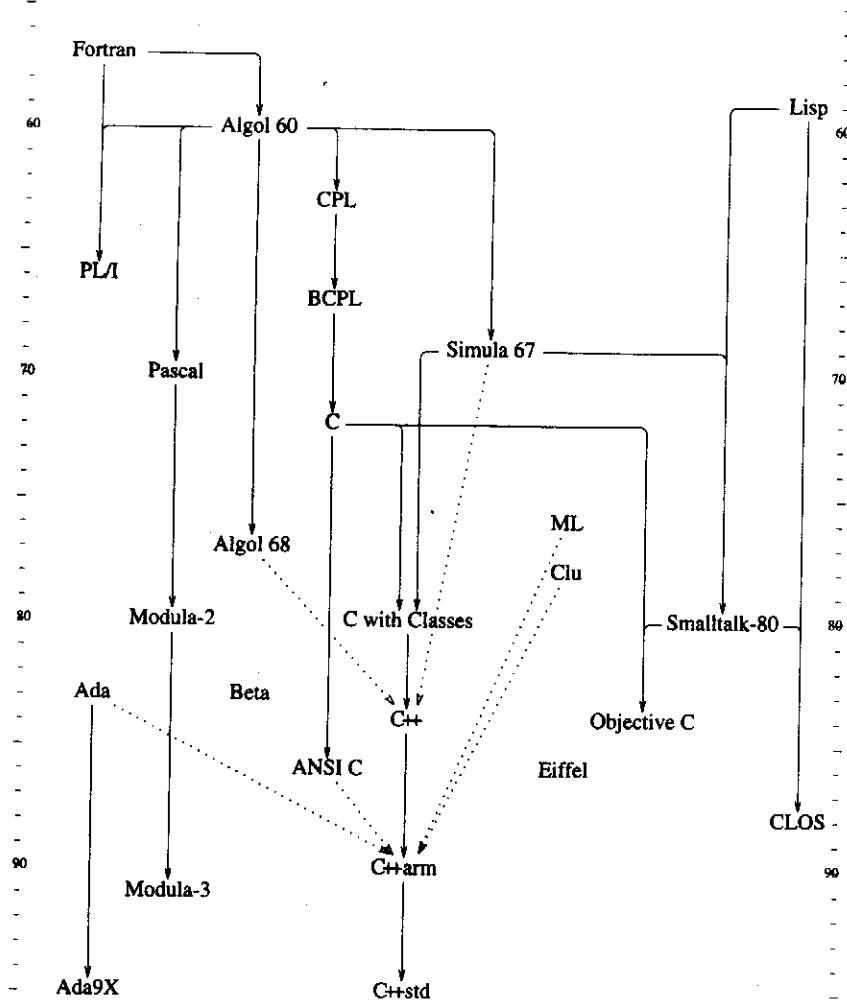
几个审稿人都要求我做一些C++语言与其他语言的比较。关于这个我已经决定不做了。在此我要重申自己长期的且强烈持有一个观点: 语言的比较很少是有意义的、更少是公平的。对于重要语言做一个很好的比较需要付出许多精力, 实际上大大超出了大部分人所愿意的付出, 超出了他们所具有的在广泛应用领域中的经验。为此还需要严格地维持一种超然的不偏不倚的观点和一种平和的理性。我没有时间, 而且作为C++的设计者, 我的不偏不倚将永远不能得到足够的信任。

我还为自己反复看到的, 在企图做语言之间公允的比较时所发生的一种现象感到忧虑。作者们常常很努力地希望能够不偏不倚, 但却毫无希望地偏向于关注某个特定的应用领域、

某种风格的程序设计或者程序员中的某种文化。更坏的是，当某种语言明显地比另外的语言更广为人知时，在看法上一种微妙转移就会发生了：这个知名语言的瑕疵被认为不那么重要了，简单的迂回处理方法被给了出来；而其他语言中类似的瑕疵却被认定是根本性的。常见的做法是，做比较或者提出指责的人根本就不知道那些不那么有名的语言里常用的迂回解决方法，因为在他们更熟悉的语言里这些方法不行。

与此类似，有关知名语言的信息总倾向于最新的，而对那些不那么有名的语言，作者依靠的常是几年以前的信息。对于那些值得去做比较的语言，拿语言X三年前定义的样子与语言Y最近试验性实现的情况去比较，这样做既不公平也无法提供有价值的信息。因此我把对C++之外其他语言的见解限制在泛义上和极特定的看法上。这是一本有关C++的书，讨论它的设计，以及促成它的演化的各种因素。这里并不试图将C++的特征与可以在其他语言里找到的东西做对照和比较。

为了把C++融进历史的大环境中，这里有一个关于许多语言第一次出现的图表，在讨论C++时常常会与这些语言不期而遇。



这个图表并不想做得尽善尽美，除了在那些对C++产生重要影响的方面。特别地，这个图对于Simula类观念的影响强调得很不够；Ada [Ichbiah, 1979] 和Clu [Liskov, 1979] 也受到了Simula [Birtwistle, 1979] 的一些影响；而Ada9X [Taft, 1992]、Beta [Madsen, 1993]、Eiffel [Meyer, 1988] 和Modula-3 [Nelson, 1991] 受到了很大影响。C++对其他语言的影响也搁在一旁没有提。图中的实线指出的是在语言结构方面的影响；虚线表示在一些特征上的影响。再多加一些线，说明每个语言间的各种关系将会使这个图变得太难看，而不可能会更有用。语言的时间指明了第一个能用的实现出现的时间。例如，Algol 68 [Woodward, 1974] 画的是1977，而不是1968。

我从对于我的HOLP-2文章的极其发散的评论里——还有其他许多来源——得到的一个结论是：对于一个程序设计语言实际上是什么，它被认定的主要用途是什么都不存在某种一致的意见。程序设计语言是一种指挥机器的工具？一种程序员之间交流的方式？一种表述高层设计的媒介？一种算法的记号？一种表述观念间关系的方式？一种试验工具？一种控制计算机化的设备的途径？我的观点是，一个通用程序设计语言必须同时是所有的这些东西，这样才能服务于它缤纷繁杂的用户集合。但也有惟一的一种东西，语言绝不能是——这也将使它无法生存——它不能仅仅是一些“精巧”特征的汇集。

在这里，观点的不同实际上反映了有关计算机科学是什么，以及语言应该如何设计等方面许多不同看法。计算机科学应该是数学的一个分支？或者工程的？或者建筑学的？或者艺术的？或者生物学的？或者社会学的？或者哲学的？换个说法，它是否从所有这些领域中借用了某些技术或者方法？我正是这样认为的。

这也就意味着语言的设计已经脱离了“更纯粹的”和更抽象的学科，例如数学和哲学。为了更好地为它的用户服务，一种通用程序设计语言必须是折衷主义的，需要考虑到许多实践性的和社会性的因素。特别地，每种语言的设计都是为了解决一个特定问题集合里的问题，在某个特定的时期，依据某个特定人群对问题的理解。由此产生了初始的设计。而后它逐渐成长去满足新的要求，反映对问题以及对解决它们的工具和技术的新理解。这个观点是实际的，然而也不是无原则的。我始终不渝的信念是，所有成功的语言都是逐渐成长起来的，而不是仅根据某个第一原则设计出来的。原则是第一个设计的基础，也指导着语言的进一步演化。但无论如何，即使原则本身也同样是会发展的。

目 录

出版者的话
专家指导委员会
译者序
前言
致读者
引言
怎样读这本书
C++ 时间表
关注使用和用户们
程序设计语言

第一部分

第1章 C++的史前时代	2
1.1 Simula和分布式系统	2
1.2 C与系统程序设计	4
1.3 一般性的背景	4
第2章 带类的C	7
2.1 带类的C的诞生	7
2.2 特征概览	8
2.3 类	9
2.4 运行时的效率	11
2.5 连接模型	13
2.5.1 纯朴的实现	15
2.5.2 对象布局模型	16
2.6 静态类型检查	17
2.6.1 窄转换	18
2.6.2 警告的使用	19
2.7 为什么用C	20
2.8 语法问题	22
2.8.1 C声明的语法	22
2.8.2 结构标志与类型名	23
2.8.3 语法的重要性	24
2.9 派生类	25
2.9.1 没有虚函数时的多态性	25
2.9.2 没有模板时的容器类	26

2.9.3 对象布局模型	27
2.9.4 回顾	28
2.10 保护模型	28
2.11 运行时的保证	31
2.11.1 构造函数与析构函数	31
2.11.2 存储分配和构造函数	31
2.11.3 调用函数和返回函数	32
2.12 次要特征	32
2.12.1 赋值的重载	33
2.12.2 默认参数	33
2.13 考虑过,但是没有提供的特征	34
2.14 工作环境	35
第3章 C++的诞生	38
3.1 从带类的C到C++	38
3.2 目标	39
3.3 Cfront	40
3.3.1 生成C	41
3.3.2 分析C++	42
3.3.3 连接问题	43
3.3.4 Cfront发布	43
3.4 语言特征	45
3.5 虚函数	45
3.5.1 对象布局模型	47
3.5.2 覆盖和虚函数匹配	49
3.5.3 基成员的遮蔽	49
3.6 重载	50
3.6.1 基本重载	51
3.6.2 成员和友元	52
3.6.3 运算符函数	54
3.6.4 效率和重载	55
3.6.5 变化和新运算符	56
3.7 引用	56
3.8 常量	59
3.9 存储管理	61
3.10 类型检查	62
3.11 次要特征	63

3.11.1 注释	63	7.1 关注和使用的爆炸性增长	120
3.11.2 建构函数的记法	63	7.1.1 C++市场的缺位	121
3.11.3 量化	64	7.1.2 会议	121
3.11.4 全局变量的初始化	65	7.1.3 杂志和书籍	122
3.11.5 声明语句	67	7.1.4 编译程序	122
3.12 与经典C的关系	69	7.1.5 工具和环境	123
3.13 语言设计的工具	71	7.2 C++的教与学	124
3.14 《C++程序设计语言》(第1版)	73	7.3 用户和应用	128
3.15 有关“什么是”的论文	73	7.3.1 早期用户	129
第4章 C++语言设计规则	76	7.3.2 后来的用户	129
4.1 规则和原理	76	7.4 商业竞争	130
4.2 一般性规则	77	7.4.1 传统语言	130
4.3 设计支持规则	80	7.4.2 更新一些的语言	131
4.4 语言的技术性规则	82	7.4.3 期望和看法	132
4.5 低级程序设计支持规则	85	第8章 库	134
4.6 结束语	86	8.1 引言	134
第5章 1985—1993年表	87	8.2 C++库设计	134
5.1 引言	87	8.2.1 库设计的折衷	135
5.2 Release 2.0	87	8.2.2 语言特征和库的构造	135
5.3 《带标注的C++参考手册》	90	8.2.3 处理库的多样性	135
5.4 ANSI和ISO标准化	91	8.3 早期的库	136
第6章 标准化	95	8.3.1 I/O流库	137
6.1 什么是标准	95	8.3.2 并行支持	139
6.1.1 实现细节	96	8.4 其他库	142
6.1.2 现实的检查	97	8.4.1 基础库	142
6.2 委员会如何运作	97	8.4.2 持续性和数据库	143
6.3 净化	99	8.4.3 数值库	143
6.3.1 查找问题	99	8.4.4 专用库	143
6.3.2 临时量的生存期	103	8.5 一个标准库	144
6.4 扩充	106	第9章 展望	146
6.4.1 评价准则	108	9.1 引言	146
6.4.2 状况	110	9.2 回顾	146
6.4.3 好的扩充问题	111	9.2.1 C++在其预期领域取得了成功吗	147
6.4.4 一致性	112	9.2.2 C++是不是一个统一的语言	147
6.5 扩充建议实例	112	9.2.3 什么是最大失误	150
6.5.1 关键词参数	113	9.3 仅仅是一座桥梁吗	150
6.5.2 受限指针	116	9.3.1 在一个很长的时期里我们还需要这座 桥梁	151
6.5.3 字符集	117	9.3.2 如果C++是答案，那么问题是什么	151
第7章 关注和使用	120		

9.4 什么能够使C++更有效	154	11.5.4 重载->*	190
9.4.1 稳定性和标准	154	11.5.5 重载逗号运算符	190
9.4.2 教育和技术	154	11.6 给C++增加运算符	191
9.4.3 系统方面的问题	155	11.6.1 指数运算符	191
9.4.4 在文件和语法之外	156	11.6.2 用户定义运算符	193
9.4.5 小结	156	11.6.3 复合运算符	194
第二部分			
第10章 存储管理	160	11.7 枚举	195
10.1 引言	160	11.7.1 基于枚举的重载	196
10.2 将存储分配和初始化分离	161	11.7.2 布尔类型	197
10.3 数组分配	162	第12章 多重继承	198
10.4 放置	162	12.1 引言	198
10.5 存储释放问题	163	12.2 普通基类	198
10.6 存储器耗尽	166	12.3 虚基类	199
10.7 自动废料收集	167	12.4 对象布局模型	203
10.7.1 可选的废料收集	167	12.4.1 虚基布局	205
10.7.2 可选的废料收集应该是什么样的	168	12.4.2 虚基和强制	206
第11章 重载	170	12.5 方法组合	207
11.1 引言	170	12.6 有关多重继承的论战	208
11.2 重载的解析	170	12.7 委托	210
11.2.1 细粒度解析	171	12.8 重命名	211
11.2.2 歧义控制	172	12.9 基和成员初始式	213
11.2.3 空指针	175	第13章 类概念的精炼	215
11.2.4 overload关键字	176	13.1 引言	215
11.3 类型安全的连接	177	13.2 抽象类	215
11.3.1 重载和连接	178	13.2.1 为处理错误而用的抽象类	215
11.3.2 C++连接的一种实现	179	13.2.2 抽象类型	217
11.3.3 回顾	180	13.2.3 语法	218
11.4 对象的建立和复制	181	13.2.4 虚函数和建构函数	219
11.4.1 对复制的控制	182	13.3 const成员函数	221
11.4.2 对分配的控制	182	13.3.1 强制去掉const	221
11.4.3 对派生的控制	183	13.3.2 const定义的精炼	222
11.4.4 按成员复制	184	13.3.3 易变性与强制	223
11.5 记法约定	185	13.4 静态成员函数	224
11.5.1 灵巧指针	185	13.5 嵌套的类	225
11.5.2 灵巧引用	186	13.6 Inherited::	226
11.5.3 增量和减量的重载	189	13.7 放松覆盖规则	228
		13.8 多重方法	231
		13.9 保护成员	235
		13.10 改进代码生成	236

13.11 到成员的指针	236	15.10 模板的实例化	289
第14章 强制	239	15.10.1 显式的实例化	290
14.1 主要扩充	239	15.10.2 实例化点	291
14.2 运行时类型信息	239	15.10.3 专门化	296
14.2.1 问题	240	15.10.4 查找模板定义	298
14.2.2 <code>dynamic_cast</code> 运算符	241	15.11 模板的作用	299
14.2.3 RTTI的使用和误用	245	15.11.1 实现与界面的分离	300
14.2.4 为什么提供一个“危险特征”	247	15.11.2 灵活性和效率	301
14.2.5 <code>typeid()</code> 运算符	248	15.11.3 对C++其他部分的影响	301
14.2.6 对象布局模型	251	第16章 异常处理	303
14.2.7 一个例子：简单的I/O对象	252	16.1 引言	303
14.2.8 考虑过的其他选择	254	16.2 异常处理的目标	303
14.3 强制的一种新记法	257	16.3 语法	305
14.3.1 问题	257	16.4 结组	305
14.3.2 <code>static_cast</code> 运算符	259	16.5 资源管理	306
14.3.3 <code>reinterpret_cast</code> 运算符	260	16.6 唤醒与终止	309
14.3.4 <code>const_cast</code> 运算符	262	16.7 非同步事件	311
14.3.5 新风格强制的影响	263	16.8 多层传播	312
第15章 模板	266	16.9 静态检查	312
15.1 引言	266	16.10 不变式	314
15.2 模板	266	第17章 名字空间	316
15.3 类模板	269	17.1 引言	316
15.4 对模板参数的限制	271	17.2 问题	317
15.4.1 通过派生加以限制	272	17.3 解决方案的思想	318
15.4.2 通过使用加以限制	272	17.4 一个解决方案：名字空间	320
15.5 避免代码重复	273	17.4.1 有关使用名字空间的观点	321
15.6 函数模板	275	17.4.2 使名字空间投入使用	322
15.6.1 函数模板参数的推断	275	17.4.3 名字空间的别名	323
15.6.2 描述函数模板的参数	277	17.4.4 利用名字空间管理版本问题	324
15.6.3 函数模板的重载	278	17.4.5 细节	326
15.7 语法	280	17.5 对于类的影响	331
15.8 组合技术	282	17.5.1 派生类	331
15.8.1 表达实现的策略	283	17.5.2 使用基类	332
15.8.2 描述顺序关系	283	17.5.3 清除全局的 <code>static</code>	333
15.9 模板类之间的关系	285	17.6 与C语言的兼容性	334
15.9.1 继承关系	285	第18章 C语言预处理器	336
15.9.2 转换	287	参考文献	339
15.9.3 成员模板	288	索引	347

第一部分

第一部分记述C++的由来，以及它从带类的C到Release 1.0的发展。这里还阐释了在这个阶段中以及后来指导着C++成长的一些规则，提供了Release 1.0之后语言开发的编年史，叙述了C++标准化的努力情况。为了提供一幅透视图，在这里也讨论了C++的使用。最后是回顾和对未来的一些思考。

章目录

- 第1章 C++的史前时代
- 第2章 带类的C
- 第3章 C++的诞生
- 第4章 C++语言的设计规则
- 第5章 编年史 1985—1993
- 第6章 标准化
- 第7章 影响和使用
- 第8章 库
- 第9章 展望未来

第1章 C++的史前时代

在过去的日子里，
邪恶当道！

——Kristen Nygaard^Θ

Simula和分布式系统——C和系统程序设计——数学、历史、哲学和文学的影响

1.1 Simula和分布式系统

C++的史前时代非常重要——在那些年里，将类似Simula的特征加进C语言的想法还没有出现在我的头脑中。但也正是在那个时期，后来造就出C++的一些准则和思想开始逐渐浮现出来。我当时是在英国剑桥大学计算实验室做博士论文，工作的目标是研究分布式系统的系统软件组织方式的其他可能途径。有关的概念框架得到高性能的（capability-based）剑桥CAP计算机及其试验性的、一直在发展中的操作系统 [Wilkes, 1979] 的支持。这个工作的细节及其结果[Stroustrup, 1979]与C++并没有太大关系。有关的是，当时我把注意力主要集中在如何用隔离良好的模块组合为软件，所用的主要实验工具是我写的一个相当大的细节繁杂的模拟器，用它模拟在分布式系统上软件的运行。

这个模拟器的初始版本是用Simula写的[Birtwistle, 1979]，运行在剑桥大学计算机中心的IBM/360 165主机上。写这个模拟器是件很令人愉快的事情，Simula的特征对于这种用途非常理想，语言提供的概念对我思考自己所面对的应用问题很有帮助，这一点给我留下了深刻的印象。类的概念使我能把应用中的概念直接映射到语言结构，使我的代码比我见过的其他任何语言的代码更具可读性。Simula的类能以协程（co-routine）的方式活动，这就使我很容易清楚地表述应用中内在的并发性。例如，很容易要求computer类的一个对象和该类的其他对象以伪并行（pseudo-parallel）的方式工作。类的层次结构可用于表述应用中的各种分层概念。例如，不同类型的计算机可以表述为类computer的各种派生类，模块间的各种通信机制可以表述为类IPC的不同派生类。在这个工作中类分层结构的使用并不很多，使用类描述并发性在我的模拟器的组织中更重要一些。

在写程序和初始排错的工作中，我对Simula类型系统的表达能力和它的编译系统捕捉类型错误的能力非常钦佩。我发现，类型错误几乎总是反映出两种情况：或者是愚蠢的编程错误，或者是设计中的概念缺陷。后者当然是一种更重要的帮助，在使用其他更原始的“强”类型系统时我从来都没有感受过这种帮助。相反，我甚至发现Pascal的类型系统比没有还要坏——它是一种枷衣，所产生的问题比解决得更多。它迫使我去扭曲自己的设计，以适应一个面向实现的人造物品。我感受到了Pascal的僵硬和Simula的灵活性，这种对比是后来开发C++的基础。我把Simula的类概念看作是最关键的差异，从那时起我就把类看作程序设计中最需要关注的问题了。

我原来就用过Simula（在丹麦Aarhus大学学习时），但这时仍然很惊喜地看到，随着程序规模扩大，Simula语言的机制也变得更有帮助了。类和协程机制，广泛而深入的类型检查保证了问题和错误不会随程序的规模而非线性地增长（如我所猜测的那样，我想大部分人也这

^Θ Kristen Nygaard是Simula语言的设计者。——译者注

样想)。相反,整个程序的活动更像是许多很小的程序的组合,而不像一个整体的大程序,因而就更容易写,更容易理解,也更容易排除其中的错误。

然而Simula的实现就完全不是同一回事了。结果使整个项目几乎变成了一场大灾难。我在那时的结论是,Simula的实现(与Simula语言相对立)实际上只是为小程序而打造的,它从根本上就不适合大程序[Stroustrup, 1979]。将分别编译的类连接起来所需要的时间完全是莫名其妙的,先编译程序的三十分之一,而后将它与程序已经编译过的其他部分连接起来,所花的时间比一下子完成整个编译和连接还要长。对于这种情况,我相信问题更多是在于主机的连接系统而不是Simula本身,但它仍然是一个障碍。在此之上的运行性能是如此之低,以至于根本无法从模拟器得到什么有用的数据。这种糟糕的运行性能应该是语言及其实现的责任,而不是应用的责任。所有这些问题对于Simula都是根本性的和无法修缮的。高代价来自一些最基本的语句特征和它们的相互作用:运行中的类型检查,变量的初始化保证,对并发的支持,对于用户创建对象和过程活动记录所做的废料收集。例如,测试数据说明,超过80%的时间被花在废料收集上,虽然这个模拟系统实际上有自己的资源管理器,因此根本就不会产生废料。今天(已是15年之后)的Simula实现已经好得多了,但是其运行性能仍然没有实现数量级的提高(根据我的了解)。

为了不终止这个项目——以至拿不到博士学位而离开剑桥——我用BCPL重写了这个模拟器,并在那个实验性的CAP计算机上运行。在BCPL[Rechards, 1980]里写代码、排除程序错误的亲身经历真是令人毛骨悚然。与BCPL相比,C就是一种非常高级的语言了,BCPL没有提供任何类型检查机制,没有任何运行时的支持。当然,作为结果的模拟器运行得确实快多了,给出了大量有用的结果,澄清了我的许多问题,也使我写出了几篇有关操作系统的论文[Stroustrup, 1978, 1979b, 1980]。

在离开剑桥时我发誓,在没有合适工具的情况下绝不去冲击一个问题,就像我在设计和实现模拟器时所遭遇的那样。这对于C++也非常重要,因为我有了一个概念:对于像写一个模拟器、一个操作系统,或者类似的系统程序设计工作这样的项目,什么样的东西才能算是一个“合适的工具”:

(1) 好的工具应该具有Simula那样的对程序组织的支持——也就是说,类,某种形式的类分层结构,对并发的某种形式的支持,以及对基于类的类型系统的强(也就是说,静态)检查。这就是我当年认识到的(今天仍然继续这样认识的)在发明程序的过程中所需要的支撑,是对设计程序(而不仅是实现程序)的支持。

(2) 好的工具产生的程序应该能运行得像BCPL一样快,在把通过分别编译得到的程序单元组合成整个程序方面也应该像BCPL那样简单而有效。如果需要把用几种语言,例如C、Algol 68、Fortran、BCPL、汇编等,写成的单元组合成一个完整的程序,某种简单的连接规则是极端重要的,这就可以使程序员避免被某一种语言的内在弱点所束缚。

(3) 好的工具应该允许高度可移植的实现。我的经验是,我所需要的“好”实现总是要等到“下一年”才能使用,而且是在一种我无法负担的计算机上。这意味着一种好工具必须有多个实现来源(没有垄断,也就是充分尊重了那些使用“不常见的”机器的用户,或者没钱的研究生们),移植时不需要复杂的运行支持系统,在工具和它的宿主操作系统之间应该只有非常有限的集成。

在我离开剑桥时这些评价准则还没有完全形成。某些东西后来成熟起来,正是反映了我

对自己在模拟器上以及在后来几年里写程序中得到的经验，以及通过讨论和阅读代码得到的其他人的经验。到了Release 2.0时C++才严格地遵循了这些准则；为C++设计模板和异常处理机制的基本压力，也源自需要校正某些违背这些准则的问题。我认为，这些准则中最重要的东西，在于它们几乎与程序设计语言的任何特定特征都没什么关系。相反，它们只是提出了对解决方案的一些约束条件。

我在剑桥期间，那里的计算实验室由Maurice Wilkes领导。对我的主要技术指导来自我的导师David Wheeler以及Roger Needham。我在操作系统领域的知识基础，以及对于模块化和通信的兴趣对于C++有持久的影响。例如C++的保护模型就来自于访问权限许可和转让的概念；初始化和赋值的区分来自于对转让能力的思考；C++的const概念是从读写保护机制中演化出来的；而C++异常处理机制的设计则受到Newcastle大学Brian Randell小组在20世纪70年代有关容错系统工作的影响。

1.2 C与系统程序设计

在1975年我曾简单地接触过C，并为它在与同类的其他系统程序设计语言、机器语言、汇编语言的比较中的表现而对它非常欣赏。关于这类语言，我熟悉PL360、Coral、Mary以及其他一些东西，而对这种语言的经验主要还是在BCPL。除了作为BCPL的用户外，我还用微程序设计做过它的中间代码形式（O-代码），实现了BCPL，因此对于这类语言的底层效率的意义有深入的理解。

在剑桥完成博士论文之后，我在贝尔实验室找到了一份工作，而后用[Kernighan, 1978]（重新）学习了C语言。所以在那个时候我不是一个C语言专家，不过是把C看成系统程序设计语言的一个最时髦和最卓越的例子。只是到了后来，我基于个人经验以及与诸如Stu Feldman、Steve Johnson、Brian Kernighan和Dennis Ritchie的讨论，才对C语言有了更深入的理解。除了C语言的特殊语言技术细节外，其中有关系统程序设计语言的普遍性思想对C++的成长至少也产生了同样深刻的影响。

我在剑桥参加的一个小项目中使用过Algol 68，因此对它了解得很多。我非常关注它的结构与C中对应概念的关系。我有时发现，如果把C的结构看成Algol 68中更具普遍意义的概念的特殊情况，对人可能很有帮助。很奇怪，我始终没把Algol 68看成一种系统程序设计语言（虽然曾经使用过一种用Algol 68写的操作系统）。我认为自己强调的是可移植性，容易与用其他语言写的代码连接，以及运行效率。我曾经多次把自己所梦想的语言描述为一个带有Simula那样的类的Algol 68。然而，就构造一个实际工具而言，选择C看起来比选择Algol 68更合适些。

1.3 一般性的背景

人们常说，一个系统的结构反映了创建它的那个组织的结构。在合理的范围内我也赞成这种看法。随之而来的是，当一个系统基本上是一个人的工作时，它就应该反映这个人的个人观点。回顾历史，我认为我个人的一般性“世界观”对C++语言整体结构的塑造产生了很大影响，其作用与造就它的计算机科学概念相当，正是这些概念构成了这个语言的各个部分。

我原来学习纯数学和应用数学，因此我在丹麦的“硕士学位”（Cand.Scient学位）是在数

学和计算机科学领域。这使我非常崇尚数学之美，但也带着一种将数学作为解决实际问题的工具的倾向，而不想将它作为一种抽象的真与美的没有明确用途的纪念碑。学生欧几里德因为提问：“那么数学又是干什么用的呢？”而被驱逐，我非常同情他。与此类似，我对计算机和程序设计语言的兴趣也完全是务实的。计算机和程序设计语言可以被当作一种艺术性的工作，但审美主义因素应该是去辅佐或提升其有用性，而不是取代或损伤它。

我的长期（持续了至少25年）爱好是历史。在大学里和毕业以后我还花了许多时间研究哲学。对于究竟应该把理性的同情放在哪里以及为什么，这些学习给了我一种非常自觉的观念。经过这样长时期的思考训练，较之理想主义者而言，我觉得自己更喜欢实用主义者（对神秘主义我更是无法赞成）。因此，我喜欢亚里士多德胜过柏拉图，休姆^Θ 胜过笛卡儿，对帕斯卡我只能感到可悲。我发现像柏拉图或者康德的那种宽泛完整的“系统”是非常奇妙的，但对它们却完全不能满意，因为它们看起来是非常危险地远离我们的日常经验和个人的基本特性。

我发现了齐克果^Θ 对个人的几乎狂热的关心以及敏锐的心理洞察力，这比黑格尔和马克思的抽象的宏伟蓝图和对人性的关心更具感染力。尊重人群而不尊重人群中的个体实际上就是什么也不尊重。C++的许多设计决策根源于我对强迫人按某种特定方式行事的极度厌恶。在历史上，一些最坏的灾难就起因于理想主义者们试图强迫人们“做某些对他们最好的事情”。这种理想主义不仅导致了对无辜受害者的伤害，也迷惑和腐化了施展权利的理想主义者们。我还发现，对于与其教义或理论出现不寻常的冲突的经验和实验，理想主义者往往有忽略它们的倾向。在理想出现问题的地方，甚至当空谈家也要赞成的时候，我宁愿提供一些支持，给程序员以选择的权利。

对文学的热爱更增强了我的认识：仅根据理论和逻辑做决策是没有希望的。在这个意义上说，C++由小说家和散文家那里得到的东西也很多，例如马丁·汉森^Θ、阿尔伯特·加缪^Θ 以及乔治·奥威尔^Θ 等。他们根本没有见过计算机，但对于C++的贡献却与计算机科学家如David Gries, Don Knuth, Roger Needham一样大。经常地，如果我试图去取缔一个我个人不喜欢的语言特征时，我总抑制住自己这样做的欲望，因为我不认为自己有权把个人观点强加给别人。我知道通过强力地推行逻辑，毫无同情心地谴责“思想中坏的、过时的、混乱的习惯”，是可能在相对短的时间里有更多的建树。但是，人的代价总是最高的。不同的人们确实会按不同的方式思考，喜欢按不同的方式做事情，对于这些情况的高度容忍和接受是我最愿意的事情。

我的希望是慢慢地——经常是令人痛苦的慢——推动人们去试验新的技术，接受那些适合他们需要或者是口味的东西。确实存在着更有效的技术去达到“宗教信仰转变”或者“革命”，但是我极端厌恶这类技术，从根本上怀疑它们在长时期和大范围上的作用。经常的情况是，如果一个人可以很容易地转变到“信仰”X，那么进一步转变到“信仰”Y也是很可能的。收获是短暂的。我喜欢怀疑论者而不是“真诚的信徒”。我把一点点实在的证据看得比许多理论更有价值，把实际经验结果看得比许多逻辑论述更重要。

Θ David Hume, 1711—1776, 苏格兰哲学家和历史学家。——译者注

Θ S. Kierkegaard, 1813—1855, 丹麦哲学家和神学家。——译者注

Θ Martin A. Hansen, 1909—1955, 丹麦小说家和散文作家。——译者注

Θ Albert Camus, 1913—1960, 法国小说家、散文家和剧作家，1957年获诺贝尔文学奖。——译者注

Θ George Orwell, 1903—1950, 英国作家和社会批评家。——译者注

这些观点也很容易走向宿命地接受现状。此外，一个人如果不打破几个鸡蛋是做不出鸡蛋饼的，而大部分人实际上确实不希望变化——至少“不是在现在”，不是以某种可能搅乱了他们日常生活的方式。这就是需要尊重事实、需要一点理想主义出现的地方。在程序设计领域里，一般地说在世界上，事情并不总处在很好的状态，要改进它们，有许多事情是可以做的。我设计C++是为了解决一个问题，而不是想证明一种观点，而它的成长又能够服务于它的使用者。这里的基本观点是，完全可能通过逐步改变去达到一种进步。最理想的情景是保持最大的变化速率，而这种变化又确实增加了它所涉及的个人的福祉。最主要的困难在于确定是什么真正构成了进步，开发出一些技术以实现平滑的转变，还要避免由于过度狂热而导致的暴行。

我愿意努力工作，采纳那些我确信能够对人有所帮助的想法。事实上，我认为，科学家和知识分子的责任就是保证他们的思想可以被公众接受，从而对社会有用，而不是为了做出一些专家们的玩物。当然，我并不想让人作为思想的牺牲品。特别地，我绝不想通过一种有局限性的程序设计语言定义去推行某种惟一的设计理念。人们思维的方式是如此的丰富多彩，企图推行一种单一理念总是弊多于利。这样，C++被有意地设计成能够支持各种各样的风格，而不是强调“一条真理之路”。

第4章论述了指导C++设计的更多细节和实际规则。在那些规则里，你可以发现上面所谈到的普遍性思想和理想的回响。

一种程序设计语言可能成为程序员日常生活中最重要的一个因素。但是无论如何，一个程序设计语言只是这个世界中微乎其微的一个部分，因此也不应该把它看得太重了。要保持一种平衡的心态，特别重要的是应该维持自己的幽默感。在各种重要的程序设计语言中，C++是俏皮话和玩笑最丰富的源泉，这并不是偶然的。

哲学性的讨论，比如有关语言特征的讨论，总倾向于过分严肃，富有说教意味。对于这些我也很遗憾，但是我仍然愿意感谢自己智慧的根源，且相信这些是无害的——好吧，至少几乎是无害的。啊不，我在文学方面的偏爱并不只限于那些强调哲理性和社会论题的作家，但他们确实在C++的丰富色彩中留下了最明显的痕迹。

第2章 带类的C

详细说明是为了对付小人。

——R. A. Heinlein

C++的直接前驱，带类的C——关键设计原则——类——运行的时间与空间效率——连接模型——静态（强）类型检查——为什么用C——语法问题——派生类——没有虚函数和模板的日子——访问控制机制——建构函数与析构函数——我的工作环境

2.1 带类的C的诞生

最终导致C++诞生的工作开始于我们企图去分析UNIX的内核，设法确定怎样才能把它分布到由局域网连接起来的一个计算机网络上。这个工作开始于1979年4月，在新泽西州Murray Hill的贝尔实验室计算科学研究中心。有两个子问题很快就浮现出来了：怎样分析由于内核分布而造成的网络流量，怎样将内核模块化。这两个问题都要求提供一种描述方式，以便描述复杂系统的模块结构和模块间的通信模式。这正好就是我曾经决定，如果没有合适工具就绝不会再碰的那一类问题。因此我就决定根据自己在剑桥形成的一套准则去开发一种合适的工具。

在1979年10月我完成了一个可以运行的预处理器，称为Cpre，它为C加上了类似Simula的类机制。到了1980年3月，这个预处理器已经得到很大的改进，能够支持一个实际项目和若干试验。按照我的记录，先后有16个项目使用了这个预处理器。第一个关键性的C++库和一个能够支持一种协程方式程序设计的作业系统[Stroustrup, 1980b] [Stroustrup, 1987b] [Shopiro, 1987]，对这些项目都是十分关键的。由这个预处理器所接受的语言被称为“带类的C”。

在从4月到10月的这个阶段里，从思考一个工具向思考一种语言的转变也开始了。但是，带类的C仍然基本上被看作是为了描述模块化和并发而做的一种C语言扩充。当然，这时已经做出了一个关键性的决定：虽然支持并发和Simula风格的模拟是设计带类的C的根本目的，但是在这个语言里并没有任何描述并发的原语。与此相反，这里是采用两种机制的结合来书写支持所需要的并行风格的库，其一是通过（类分层中）继承性，其二是通过定义可以由预处理器识别的具有特殊意义的类成员函数。请注意，在这里的“风格”用的是复数，我当时认为这是至关重要的，现在仍然持同样观点：这个语言中应该能够表述多种并行性的概念。实际中存在着许多应用，对它们而言并行性的支持是必需的。但在另一方面，并不存在一种处于主导地位的并行模型。这样，当需要这类支持时，就应该通过库或者特定的扩充来做这件事，只有这样，才不会出现一种特定形式的并行支持排斥其他形式的问题。

这个语言就这样提供了对程序组织的一般性机制，但又不去支持特定的应用领域。正是这一点使得带类的C——以及后来的C++——成为一种通用的程序设计语言，而不是变成为支持某类特殊应用而扩充出的C语言变形。后来，在是为特殊应用提供支持，还是提供通用抽象机制之间做选择的情况又一再出现，而所做的每次决策都改进了抽象机制。这样C++就没有

提供内部的复数、字符串或者矩阵类型，也没有对并行性、持久性（*persistence*）、分布式计算、模式匹配、文件系统操作等提供直接的支持。这些不过是最经常提出的许多扩展要求中的几个。支持这些功能的库当然都已经存在了。

对带类的C的早期描述作为贝尔实验室技术报告发表在1980年4月[Stroustrup, 1980]以及SIGPLAN Notices[Stroustrup, 1982]。更详细的贝尔实验室技术报告——“给C语言增加类功能：语言演化的一个练习”（*Adding Classes to the C Language: An Exercise in Language Evolution*）[Stroustrup, 1982]发表在《软件：实践和经验》杂志。这些文章是一个例子，在其中只描述了那些已经完全实现并且已经被使用的特征。这也是贝尔实验室计算科学研究中心的传统。这种政策后来有所改变，因为确实需要给C++的未来提供更多的开放性，这是为了保证C++的许多非AT&T用户能够对C++的发展进行一场自由而开放的辩论。

很明显，带类的C的设计能允许人们更好地去组织程序，“计算”仍被看作是需要C语言解决的问题。我当时很担心，害怕不能在运行方面的代价与C语言相近的情况下真正改善程序的结构。当时最明确的目标就是在运行时间、代码紧凑性和数据的紧凑性方面能够与C相比美。例如，有一次某人证明，由于带类的C预处理器在函数返回机制中不适当当地引进了一个临时变量，导致程序的整体运行效率（与C相比）出现了3%的下降。这是完全不能接受的，这个额外开销被清除掉了。与此类似，为了保证在数据布局方面与C的兼容性，也为了避免额外的空间代价，在类对象中没有放任何“内务数据（*housekeeping data*）”。

那时最关注的另一个问题就是避免带类的C存在使用领域方面的限制。基本设想是——后来也确实做到了——带类的C应该能用到C可以使用的一切地方。这还意味着需要取得与C相当的执行效率，带类的C不应该为了去掉C语言的“危险”或“丑陋”特性而付出效益方面的代价。我不得不经常对人们重复这种观点/原则（很少是对带类的C的用户），这些人希望能够把带类的C弄得更安全些，所建议的方式就是增加类似早期Pascal那样的静态类型检查。另一种提供“安全性”的方法是为所有可能不安全的操作增加运行中的检查，对于排错环境（*debugging environment*）而言这是很合理的，但是语言不可能在保证这种检查的同时又不丢掉C语言在运行时间和空间效率方面的最大优势。所以带类的C就没有提供这些检查，虽然某些C++环境为了排错而提供了这类检查。当然，如果用户需要并且也负担得起，可以自己加入运行时的各种检查（参见16.10节和[2nd]）。

C语言提供了许多低级操作，例如位操作和在不同大小的整数中做选择。在这里还存在着许多机制，例如显式的不加检查的类型转换可以用于故意突破基本类型系统。带类的C和后来的C++仍然追寻着同一条路，维持了C的低级操作和不安全特征。与C不同的是，C++系统化地清除了使用这些操作的必要性，除了在那些必须使用它们的地方，而且只是在程序员明确要求时才使用不安全操作。我当时强烈地感到（现在依然如此），在写每个程序时都不存在某种惟一的正确途径，而作为程序设计语言的设计者，也没有理由去强迫程序员使用某种特定的风格。但是在另一方面，他们也确实有义务去鼓励和支持各种各样的风格和实践，只要那些东西已经被证明是有效的。他们还应该提供适当的语言特性和工具，以帮助程序员避免公认的圈套和陷阱。

2.2 特征概览

在1980年初的实现中提供的特征总结如下：

- 1) 类 (2.3节)
- 2) 派生类 (但是还没有虚函数, 2.9节)
- 3) 公用/私用[⊖] 的访问控制 (2.10节)
- 4) 建构函数和析构函数 (2.11.1节)
- 5) 调用和返回函数 (后来删除了, 2.11.3节)
- 6) friend类 (2.10节)
- 7) 函数参数的检查和类型转换 (2.6节)
- 在1981年里又加入了三个特征:
- 8) 在线函数 (2.4.1节)
- 9) 默认参数 (2.12.2节)
- 10) 赋值运算符的重载 (2.12.1节)

因为带类的C是通过一个预处理程序实现的, 因此只需要描述新特征 (也就是说, 那些C语言里没有的特征), C 的全部能力都是用户可以用的。这两个方面在那时都受到了称赞。以C作为子集极大地减少了所需要的技术支持和文档的重写工作。这也是非常重要的, 因为在一些年里我除了做试验、设计和实现外, 还要完成所有有关带类的C和C++的文档和技术支持。由于C的所有特征都能用, 这就保证了不会由于我这方面的偏见或者缺乏远见而引进了某些限制, 使用户不能享有某些原来在C中已有的特性。自然, 移植到所有支持C语言的机器上也是有保证的。开始时, 带类的C在一台DEC的PDP/11上实现和使用, 不久它就被移植到许多机器上, 例如DEC VAX和基于Motorola 68000的机器等。

带类的C当时仍然被看作是C的一种方言, 而不是另一个语言。进而, 类也被说成是“一种抽象数据类型机制” [Stroustrup, 1980]。当时还没有提出对面向对象程序设计的支持, 这种情况一直延续到1983年, 在那时C++提供了虚函数 [Stroustrup, 1984]。

2.3 类

很清楚, 带类的C最重要的方面——后来的C++也一样——就是类的概念。带类的C中的类概念的许多方面都可以从下面的简单例子中看到 [Stroustrup, 1980][⊖] :

```
class stack {
    char s[SIZE]; /* array of characters */
    char* min;     /* pointer to bottom of stack */
    char* top;     /* pointer to top of stack */
    char* max;     /* pointer to top of allocated space */
    void new();    /* initialize function (constructor) */
public:
    void push(char);
    char pop();
};
```

-
- ⊖ public/private, 一般被译为共有或公用/私有。这两个词描述的是访问控制问题: 某种功能是/否提供给外界使用, 因此是关于使用权, 而不是所有权 (所有权非常清楚, 根本无须讨论)。据此, 本书中将它们一律翻译为“公用”和“私用”, 这样更符合原意。——译者注
 - ⊖ 在这些例子里, 我保留了带类的C原来的语法和风格。与C++和现在风格不同的地方不会对理解造成任何问题, 但可能有些读者对这一点感兴趣。我已经 (无论如何) 修正了某些明显的错误, 并添加了注释作为补充, 这是原来的正文中所没有的。

类是用户定义的数据类型，它刻画了类成员的类型，定义了这种类型的变量（这个类的对象）的表示形式，定义了一组处理这些对象的操作（函数），以及这个类的用户对这些成员的访问方式。成员函数通常在“其他地方”定义：

```
char stack.pop()
{
    if (top <= min) error("stack underflow");
    return * (--top);
}
```

现在就可以定义类stack的对象了：

```
class stack s1, s2;           /* two stack variables */
class stack * p1 = &s2;        /* p1 points to s2 */
class stack * p2 = new stack; /* p2 points to stack object
                             allocated on free store */

s1.push('h');    /* use object directly */
p1->push('s'); /* use object through pointer */
```

从这里反映出的关键设计决策包括：

(1) 带类的C遵循Simula的方式，让程序员去描述类型，变量（对象）通过这些类型来建立。这里没有采用例如Modula一类的方式，在那里描述的是汇集了对象和函数的模块。在带类的C中（C++也是一样），一个类就是一个类型（2.9节），这是C++里最关键的概念。既然在C++里的class意味着用户定义类型，为什么我不直接称它为type呢？选择用class这个词的基本原因是我不想发明新术语。此外，我也觉得对于大多数情况而言，Simula的术语都是很合适的。

(2) 在这里把用户定义类型的对象的表示作为类定义的一个部分。这样做带来了非常深远的影响（2.4节，2.5节）。例如，这就意味着不一定要使用自由存储区（也被称为堆存储，或者动态存储）以及废料收集机制，能够实现用户定义类型的真正的局部变量。这也意味着，如果一个函数中直接使用的某个对象的表示形式改变了，这个函数就必须重新编译。参见13.2节中有关的C++机制，在那里可以看到如何通过描述界面的方式避免这种重新编译。

(3) 采用编译时的访问控制限制对实际表示的访问。按照默认的方式，只有在类声明中给出的函数才能使用类成员的名字（2.10节）。在公用界面中描述的成员（写在public后面的那些声明，通常是函数成员）可供其他代码使用。

(4) 对于函数成员要描述其完整的类型（既包括返回类型，也包括参数类型）。静态（编译时）的类型检查需要这种类型规范描述（2.6节）。这一点也与那时的C不同。在那个时候C的界面并不要求描述参数类型，调用时也不做任何检查。

(5) 函数定义通常被写在“其他地方”，以使类的声明看起来更像一个界面描述，而不像是为了组织代码而提供的一种语法结构。这也意味着更容易进行分别编译，传统的为C而使用的连接技术就足以支持C++（2.5节）。

(6) 函数new()是建构函数，这个函数对于编译系统具有特殊意义。这种函数为类提供了一种保证（2.11节）。这里的有关保证是指建构函数（虽然那时称它为new函数容易使人感到疑惑）一定会被调用，以初始化它所属的类的每个对象，这项工作一定要在第一次使用对象之前完成。

(7) 同时提供了指针和非指针类型（就像C和Simula一样）。在这里同时允许指向用户定义类型和内部类型的指针，这一点更像C而不像Simula。

(8) 像C语言里一样，对象的分配可以有三种方式：在堆栈上（作为自动对象），在固定地址（静态对象），或者在自由存储区（在堆，或者说是动态存储区里）。与C语言不同的是，带类的C为自由存储的分配和释放提供了特定的运算符new和delete（2.11.2节）。

带类的C和C++后来的发展，完全可以被看作是在进一步探索这些设计选择的逻辑推论，开发这些选择的好的一面，并设法弥补由它们的缺点一面所引出的问题。这些设计选择的许多（并不是全部）内涵在那时已经被认识到了，[Stroustrup, 1980]发表的日期是1980年4月3日。本小节就是想解释那时我们已经理解了什么东西，并指明在哪些小节里将解释有关的推论和后来的认识。

2.4 运行时的效率

在Simula里不能有“类类型[⊖]”的局部变量或者全局变量，也就是说，类的所有对象都必须通过new操作在自由存储区里分配。对于我在剑桥所做的模拟器的测试表明，这是低效率的一个主要根源。后来挪威计算机中心的Karel Babcík对Simula运行时的执行情况给出了一些数据，也证实了我的猜测[Babcík, 1984]。仅仅由于这个原因，我就希望能够有类类型的局部和全局变量。

此外，对内部类型和用户定义类型采用不同创建规则和作用域规则也是不精致的。由于在Simula里缺乏局部的和全局的类变量，我有时就感觉到自己的程序设计风格受到了束缚。与此类似，我也曾希望在Simula里有指向内部类型的指针，因此我希望有C的指针概念，它能够统一地作用到用户定义类型和内部类型上。这一点就是后来成长为C++设计的一条经验法则的初始概念，这个法则是：用户定义类型和内部类型与语言法则的关系应该是一样的，能够从语言及其相关工具方面得到同样程度的支持。在这个法则形成时，内部类型得到的支持要多得多，但是C++已越过了这个目标，现在内部类型得到的支持反而比用户定义类型稍微弱了一点。

在带类的C的初始版本里没提供在线（inline）函数，以从可用的表示形式中进一步获益。当然，不久我们就提供了在线函数。引进在线函数的一般性理由与越过保护屏障的代价有关，这种代价有可能导致人们不愿意用类去隐藏表示的细节。特别地，[Stroustrup, 1982b]中观察到人们总把数据成员做成公有的，以避免调用简单类的建构函数而带来的开销，因为对这些对象的初始化可能只需要一两个赋值语句。将在线函数引进带类的C，直接原因是一个具体的项目。在该项目中，由于某些类与实时处理有关，这种函数调用的开销是无法接受的。为了使类机制能够成为在这个应用中有用的东西，就要求在跨越保护屏障时不付出任何代价。只有在类声明中提供一种可用表示，并能把对公用（界面）函数的调用都变成在线的，才可能达到这个目的。

在这些年里，沿着这条线索的思考逐渐演变成一条C++设计规则：只提供一个特征是不够的，还必须以一种实际上可以负担得起的形式来提供它。在这里，可以负担得起意味着“使用常见硬件的开发者可以负担”，而不是“使用高端设备的研究者可以负担”或者“若干年之后，当硬件变得更便宜之后就可以负担”，这几乎被作为一个定义。带类的C始终被作为

[⊖] 类类型（class type），指用户定义的类所形成的类型，在本书中常有这种说法。——译者注

一种就要在当时或下一个月里使用的东西，而不是一个在几年之后有可能发布某种东西的研究项目。

在线机制

对于类的使用，在线被认为是非常重要的机制。这样，问题就不是要不要提供这种机制，而更多的是如何提供它。有两个论据在那时取得了胜利，导致了让程序员决定编译程序应该让哪些函数成为在线的方式。首先，有些语言把在线问题留给编译程序去做，因为“编译系统对事情了解得最清楚”，但是我曾有过对这种语言很糟糕的亲身体验。只有在程序中写出了要求在线，编译程序有了关于时间/空间优化的概念时，它才能够做得最好，这与我的看法相符。我对其他语言的经验是，只有“下一个版本”才能够实际地做在线工作，而且那个版本将要根据一种程序员无法有效控制的内部逻辑完成这个工作。有些情况使这个问题更困难，C（以及带类的C和后来的C++）有其固有的分别编译机制，这就使编译程序只能访问程序中的一小部分（2.5节）。要将某个函数变成在线的，而你又不知道它的源代码，这件事是不可能，但却需要有高级的连接系统和优化技术。而这种技术在当时是不存在的（至今在大部分环境里仍然不存在）。进一步说，采用全局分析一类的技术，在没有用户支持下做自动在线处理，大概无法适应很大的程序。在设计带类的C时，我们是希望在传统系统上能生成高效的代码，给出简单而容易移植的实现。提出了这些要求之后，程序员就必须提供帮助了。在今天看这个选择仍然是正确的。

在带类的C中只有成员函数能做成在线的，而要求函数成为在线只有一种方式，那就是把它的体放进类的声明之中。例如：

```
class stack {
    /* ... */
    char pop()
    {
        if (top <= min) error("stack underflow");
        return *--top;
    }
};
```

事实上，那时也看到这会使类的声明显得比较杂乱。另一方面，这看起来也是个好东西，因为它不鼓励在线函数的过度使用。关键字`inline`和允许在线非成员函数的功能都是后来由C++提供的。例如，在C++中可以写下面这样的代码：

```
class stack { // C++
    // ...
    char pop();
};

inline char stack::pop() // C++
{
    if (top <= min) error("stack underflow");
    return *--top;
}
```

这种`inline`指示字只不过是一种提示，编译系统可以去做，常常也会忽略它们。这一点在逻辑上是必需的，因为某人可能写出一个递归的在线函数，在编译时无法证明它不会导致一个无穷递归，试图将这类东西变成在线的就会引起无穷的编译。让`inline`作为一种提示在实践

中也有优势，因为这样可以使写编译系统的人更容易处理各种“病态”情况，对它们简单地不做在线处理。

带类的C要求——它的后继者也一样——在线函数在整个程序里必须具有惟一定义。如果在不同编译单元里定义了上面那样的`pop()`函数，就会因为搅乱了类型系统而引起巨大的混乱。由于分别编译的存在，要在一个大系统里保证绝不出现这种灾难性情况也是极端困难的。带类的C并没有做这种检查，大部分C++实现也还没有想去保证所有分别编译的单元里所定义的在线函数都互相不同。无论如何，这个理论问题还没有作为实际问题而浮现出来，出现这种情况，主要是由于在线函数通常都与类一起定义在头文件里，而类声明在整个程序里也需要有惟一性。

2.5 连接模型

如何将分别编译的程序片段连接在一起，这个问题对于任何程序设计语言都是非常重要的，它也在一定程度上决定了这个语言所能提供的特征。对开发带类的C和C++语言最有影响的事情之一就是有关下面问题的决策：

- 1) 分别编译应该能使用传统的C/Fortran、UNIX/DOS风格的连接程序。
- 2) 连接应该具有类型安全性。
- 3) 连接过程不应该要求任何形式的数据库（当然可以借助这种东西来改善实现）。
- 4) 应该很容易并且高效地与采用其他语言（如C、汇编或Fortran）写出的程序片段连接到一起。

C语言采用头文件保证分别编译的一致性。对数据结构布局、函数、变量和常量的声明被放置在头文件里。在典型情况下，这样的头文件被以文本形式包含进每个需要这些声明的源文件中。保证一致性的方式就是把适当的信息放入头文件，并保证能够一致地包含这些头文件。C++在一定程度上沿袭了这种模式。

布局信息可以放在C++的类声明中（虽然不必这样做，参看13.2节），这样做的原因就是为了能非常有效地声明和使用真正的局部变量。例如：

```
void f()
{
    class stack s;
    int c;
    s.push('h');
    c = s.pop();
}
```

使用2.3节和2.4节的`stack`类声明，甚至带类的C的最简单实现也能保证这个例子不使用自由存储区。这里对`pop()`的调用是在线的，不会引进函数调用的开销；非在线的`push()`调用可以使用分别编译的函数。在这个方面C++与Ada类似。

究竟是使用互相分离的界面和实现声明（就像在Modula里那样），再加一个拼配它们的工具（连接程序）；还是用一个类声明加上一个工具（依赖关系分析程序），该工具把界面部分与实现细节分开考虑，以便完成分别编译？那时我觉得可以在这两者间做一种折衷。看起来我低估了后者的复杂性，而主张前者则说明我那时低估了它的代价（无论从移植，还是从运行时间的角度看）。

那时我还把C++社团的事情弄得很糟，因为我没有正确地解释可以通过派生类来得到界面与实现的分离。我曾经试过（参见例如[Stroustrup, 1986, 7.6.2节]），但不知什么缘故我却从未把消息传过去。我想，这个失败的主要原因是我和许多（大部分？）C++程序员从没有想到过这件事。而非C++程序员则看着C++想，因为你能够把表示数据直接放进描述界面的类声明，你必须这样做。

我没有企图去为带类的C提供强制性的类型安全连接的工具，这种东西要等到C++的Release 2.0。但是，我记得与Dennis Retchie和Steve Johnson谈过，应该把越过编译边界的类型安全性看作C的一个部分。我们只不过是缺少对实际程序的强制性手段，所以才要依赖于例如Lint[Kernighan, 1984]那样的工具。

特别地，Steve Johnson和Dennis Retchie肯定地说C的意图是采用按名字等价而不是按结构等价。例如：

```
struct A { int x, y; };
struct B { int x, y; };
```

定义的是两个不相容的类型A和B。进一步说：

```
struct C { int x, y; } // in file 1
struct C { int x, y; } // in file 2
```

定义两个不同的类型，它们都叫作C，而一个能越过编译单元的边界做检查的编译程序应该给出一个“重复定义”错误。采用这个规则的原因是为了使维护问题减到最小。这种完全相同的声明通常不大会出现，除非是做的拷贝。而一旦被拷贝到另一个源文件之后，这个声明通常不会保持这种样子而不改变。当一个声明被改变的时候——而另一个却没有，程序就会非常奇怪地无法正常工作了。

作为一个实践性问题，C语言以及后来的C++都保证，如上的A和B这样的类似结构具有类似的布局，这样就可能在它们之间转换，以明显的方式使用它们：

```
extern f(struct A*);

void g(struct A* pa, struct B* pb)
{
    f(pa); /* fine */
    f(pb); /* error: A* expected */

    pa = pb; /* error: A* expected */
    pa = (struct A*)pb; /* ok: explicit conversion */

    pb->x = 1;
    if (pa->x != pb->x) error("bad implementation");
}
```

按名字等价是C++类型系统的基石，而布局相容性规则保证了可以使用显式转换，以便能提供低级的转换服务。而在其他语言里就只能通过结构等价规则提供这种东西。与结构等价规则相比，我更喜爱按名字等价，因为我认为它是最安全最清晰的等价模型。由此我也非常高兴地看到，这个决定并没有使我陷入与C语言的相容性问题，也没有把提供低级服务搞得更麻烦。

这一认识后来成长为“惟一定义规则”：在C++语言中，每个函数、变量、类型、常量等都应该恰好有一个定义。

2.5.1 纯朴的实现

要考虑一些纯朴的实现，部分地是由于在开发带类的C时极端缺乏资源，部分地是由于不信任那些要求巧妙技术的语言和机制。最早形成的有关带类的C设计目标中有一个要求是“不应该使用比线性检索更复杂的算法”。当这个粗略的规则被违背时，例如在函数重载的情况下（11.2节）——就会导致一种比较复杂的语义，使我感到不舒服。许多情况下这也导致实现的复杂化。

那时的目标——基于我在Simula上的经验——就是要设计出一种语言，它由于很容易理解而能够吸引用户，由于其实现起来也足够容易，因而也就能够吸引实现者。一种相对简单的实现必须能生成在正确性、运行速度和代码规模方面可与C代码相媲美的代码。一个相对初级的用户在一个相对缺乏帮助的程序设计环境中，应该能够用这个实现去做实际项目。只有当这两条准则都能够满足时，带类的C以及后来的C++才能在与C语言的竞争中生存下去。这个原则的一个早期叙述是：“带类的C必须是像C或者Fortran那样的野草，因为我们没能力负担起照看像Algol 68或者Simula那类玫瑰的事务。如果我们发布了一个实现，而后离开了一年，我们希望在回来时还能够看到几个正在运行的系统。如果系统需要复杂的维护，或者把它简单地移植到一台新机器上需要的时间超过一个星期，这种情景就不可能出现了。”

这就是在用户中培养满足感的某种哲学的一个主要部分。很明显，目的总是要在使用C++的各个方面开发出局部性的专业经验。多数组织必然会循着另一种策略，它们将用户与服务隔离开，由服务产生收益去支持一种集中式的支撑组织或者一批顾问，或者两者皆备。按照我的观点，这种对照正是C++和许多其他语言之间的一种本质性差异。

决定使用一种相对基本的——同时又几乎是普遍可用的——C连接框架，这就带来了一个根本性的问题：C++编译程序必须在只可以使用部分程序信息的情况下工作。关于一个程序的某个假定，明天很可能被用别的语言（例如C、Fortran或者汇编）写出后连接进来的另一个程序所破坏，而这个连接甚至可能在整个程序开始执行之后。这种问题可能在许多上下文中表现出来。要求一个实现做出如下保证是很困难的：

- 1) 某种东西具有惟一性。
- 2) 信息是一致的（特别地，类型信息是一致的）。
- 3) 某个东西已经初始化过。

此外，C语言对于名字空间的隔离只提供了最弱的支持，这样，通过分别写出各个程序片段来防止名字空间的污染就成了一个问题。在随后的年代里，C++试着去面对所有这些挑战性的问题，然而又不想偏离其基本模型和技术，因为正是这些东西支持着可移植性和执行效率。但是在带类的C的年代里，我们只能简单地依赖于C的头文件技术。

通过C连接程序的可接受性又形成了开发C++的另一条经验规则：C++只是一个系统中的一个语言，而不是一个完整的系统。换句话说，C++应该扮演传统程序设计语言的角色，与语言、操作系统或程序员世界中的其他部分有着根本性的区别。这就限制了语言所扮演的角色，不必去做那些语言不容易做的事情，不像Smalltalk和Lisp那样被认为是完整的系统或者环境。这种做法也使一个C++程序片段能够调用其他语言写出的另一个片段成为一种基本要

求；C++程序片段同样也可以被用别的语言写的程序片段所调用。“只是一个语言”还能使C++的实现很容易从其他语言写出的工具那里获益。

需要一种程序设计语言，并希望用它写出的代码能够像一台大机器里的齿轮，这些对于大多数企业界用户都是最重要的事情。能够与其他语言和系统共生的问题至今还明显不是大多数理论家、所谓的完美主义者、学术界用户们的主要关注点。而我却相信这是C++成功的一个主要原因。

带类的C与C语言几乎是代码兼容的。当然，它并不是百分之百的兼容，例如，像class或者new这样的词在C里是完全正常的标识符，但在带类的C和它的后继语言里则成了关键字。它们是连接兼容的，C的函数可以在带类的C里面调用，带类的C的函数也可以在C里调用，struct在两个语言里的布局也相同，所以在这两个语言间传递简单对象或组合对象都是简单而有效的事。这种连接兼容性也一直维持到C++（也有几个简单而明显的例外，程序员如果需要的话很容易避免它们（3.5.1节））。按照这些年我和我的同事们的经验，这种连接兼容性要比代码的兼容性重要得多。这至少说明，同样的代码在C和C++里，或者能够给出同样的结果，或者在某个语言里无法完成编译或者连接。

2.5.2 对象布局模型

从某种意义上说，对象的基本模型是带类的C设计中最基本的问题。我对于对象在存储器里应该是什么样子总持有一种很清晰的观点，也总在考虑语言特征将怎样影响到对于这种对象的操作。对象模型的演化也是C++语言演化的最基本部分。

带类的C的对象简单地就是一个C结构。这样，

```
class stack {
    char s[10];
    char* min;
    char* top;
    char* max;
    void new();
public:
    void push();
    char pop();
};
```

的布局与下面的结构完全一样

```
struct stack { /* generated C code */
    char s[10];
    char* min;
    char* top;
    char* max;
};
```

也就是

char s[10]
char* min
char* top
char* max

为了对齐，编译系统可能将一些“填充”加进成员之间或者加到最后，除去这种因素，对象的大小就是成员的大小之和了。这也使存储的使用达到了最小。

类似地，通过对成员函数的调用做直接映射，也使运行的时间开销达到了最小。

```
void stack.push(char c)
{
    if (top>max) error("stack overflow");
    *top++ = c;
}

void g(class stack* p)
{
    p->push('c');
}
```

相当于在一般代码里调用一个等价的C函数：

```
void stack__push(this,c) /* generated C code */
struct stack* this;
char c;
{
    if ((this->top)>(this->max)) error("stack overflow");
    *(this->top)++ = c;
}

void g(p) struct stack* p; /* generated C code */
{
    stack__push(p,'c');
}
```

在每个成员函数里有一个称为this的指针，它所引用的就是调用成员函数的那个对象。Stu Feldman还记得，在带类的C的最早实现里，程序员不能直接访问this。在他指出这一点之后我立刻纠正了这个问题。如果没有this或者其他与此等价的机制，成员函数将无法被用到链接表的操作中。

C++的this指针是Simula里THIS引用的翻版。有时人们会问这个问题：为什么this是一个指针而不是一个引用？为什么它叫this而不叫self？当this被引进带类的C中时，在这个语言里还根本没有引用机制。另一方面，C++是从Simula而不是Smalltalk那里借用的术语。

在把stack.push()说明为inline之后，生成的代码看起来应该是这个样子：

```
void g(p) /* generated C code */
struct stack* p;
{
    if ((p->top)>(p->max)) error("stack overflow");
    *(p->top)++ = 'c';
}
```

这应该就是程序员在C语言里写出的代码。

2.6 静态类型检查

关于如何把静态（“强”）类型检查引进带类的C，我没有任何有关讨论的记忆，没有任何

设计记录，也没有有关实现问题的记忆。带类的C在语法和规则里的那些直接由ANSI C来的东西都以完全形式出现在第一个带类的C的实现中。在此之后，不多的一些经验引出了C++目前的（更严格的）规则。在经历过Simula和Algol 68的经验之后，静态类型检查对于我而言已经是一种必需品，惟一的问题只是如何把它加进来。

为了避免排斥C代码，我决定允许调用不加声明的函数，而对这种不加声明的函数也不进行检查。这当然是类型系统里的一个大漏洞。后来还做了些努力，以设法降低它作为程序设计中错误根源的重要性。最后，在C++里这个漏洞被完全堵住了：在那里调用不加声明的函数是非法的。一个简单的观察就排除了所有的折衷企图：学过带类的C的程序员竟丧失了寻找由于简单类型错误而造成的运行错误的能力。由于他们逐渐习惯于依赖带类的C所提供的类型检查和类型转换，甚至丧失了快速地发现某些愚蠢错误的能力，而这些错误是由于缺乏检查而混进C程序的。进而他们也不能为避免这种愚蠢错误而采取预防措施，而好的C程序员则会把这些看作是理所当然的事情。总之，“这种错误根本不会出现在带类的C里”。这样，随着未检查出的参数类型错误所引起的运行错误的频率下降，它们的严重性和为排除它们而花费的时间却增加了。这种结果严重地困扰着程序员，也导致他们要求进一步收紧类型系统。

关于这种“不完全检查”最有趣的经验是一种技术：允许调用未加声明的函数，但是却注意所使用的参数类型，当再次遇到有关调用时进行一致性的检查。许多年之后，Walter Bright独立地发现这种技巧之后，他将其命名为自动原型，采用的是ANSI C对函数声明使用的术语原型。经验说明自动原型能捕捉到许多错误，在开始时将增强程序员对这个类型系统的信任。但是，如果错误本身具有一致性，或者错误出现在那些编译中只调用了一次的函数，它们就不会被捕获到。自动原型最终将破坏程序员对类型检查的信任，从而引进了一种偏执狂意识，比我在C或BCPL那里看到的情况还要糟。

带类的C引进了概念 `f(void)`，这声明 `f` 是一个没有参数的函数。与此相对的是C里的 `f()`，它声明了一个可以具有任何类型的任意个参数的函数，而且不进行任何类型检查。用户们不久就使我确信 `f(void)` 在记法上很不优雅，让 `f()` 接受参数也是不符合直觉的。最后，试验的结果是用 `f()` 表示没有参数的函数，正如许多初学者所期待的那样。由于有 Doug McIlroy 和 Dennis Ritchie 的支持，我斗胆认可了在这里与C的分离。只是在他们用了可憎的去指责 `f(void)` 之后，我才敢于去给 `f()` 规定它最明显的意义。当然，到今天为止C的类型规则还是比C++宽松得多，而且ANSI C从带类的C那里借去了“可憎的 `f(void)`”。

2.6.1 窗转换

在收紧带类的C的类型规则时，早期的另一想法就是不允许“破坏信息的”隐含转换。我也和别人一样对下面这样的例子感到震惊（这种东西在实际程序中当然很难看到）：

我决定要试着禁止所有不能保持值不变的转换，也就是说，只要想把一个大的对象存储到较小的对象里，就要求明显地写出转换运算符：

```
void g(long lng, int i) /* experiment */
{
    int il = lng; /* error: narrowing conversion */
    il = (int)lng; /* truncates for 16 bit ints */

    char c = i; /* error: narrowing conversion */
    c = (char)i; /* truncates */
}
```

这个试验失败得很惨。我检查的每个C程序都包含大量从int到char变量的赋值。很自然，因为这都是正在工作的程序，这种赋值中的绝大多数必然是安全的。也就是说，或者有关的值原本就足够小，实际上并没有真正的截断；或者有关截断就是人们所需要的，至少是在当时的上下文里不会造成损害。在带类的C的团体中，没人希望这样背离C语言。我现在还在寻找弥补这类问题的方法（14.3.5节）。

2.6.2 警告的使用

我还考虑了在运行中对所赋的值进行检查，但这样做将在时间和代码规模方面带来极大的代价，而且按照我的看法，这时再去检查问题也太迟了。这样，运行中对转换的检查——更重要的一般性检查——通常都归入到“将来由排错系统支持”的范畴。与上述方式不同，我采用另一种技术去处理C语言里那些我认为极端严重而不能忽略的弱点，但这些弱点又因为在C语言的结构里根深蒂固而无法去除。这种技术后来也变成了一种标准。我让带类的C的预处理器（以及后来的C++编译系统）发出警告：

```
void f(long lng, int i)
{
    int il = lng; // implicit conversion: warning
    il = (int)lng; // explicit conversion: no warning

    char c = i; // too common to repair: no warning
}
```

对于long->int以及double->int总是无条件地发出警告（现在也是这样），因为我看不出有什么道理说这种转换是合法的。这种转换不过是历史上某些偶然事件的简单后果，因为浮点数被引进C语言是在显式转换的引进之前。关于这类警告我没有收到过任何抱怨，我和其他人都无数次地被这种功能所拯救。对于转换int->char，我感到无法做任何事情。迄今为止这种转换还是能够通过AT&T的编译系统，没有任何警告。

在这样做时，我决定只将无条件警告的功能用在那些“具有超过90%的可能性会造成实际错误”的情况。这反映了人们在C编译系统和Lint上的经验，在这些系统中，警告在更多情况下并不是“错误”，从某种意义上说，人们是用警告去反对某些东西，虽然它们实际上并不会引起程序的错误行为。这种做法反而导致程序员们倾向于忽略C编译系统所产生的警告，或者只是很不情愿地去注意它们。我的意图则是设法保证忽略C++的警告将被看成是一种愚蠢行为。我认为我是成功了。这种警告实际上被用作一种补充手段，如果为保持与C语言的兼容性而导致某些问题无法通过修改语言来解决，那么就使用这种手段。这也是一条路，它能使从C语言到C++的转换

变得更容易些。例如：

```

class X {
    // ...
}

g(int i, int x, int j)
    // warning: class X defined as return type for g()
    // (did you forget a ';' after ')' ?)
    // warning: j not used
{
    if (i == 7) { // warning: constant assignment
        // in condition
        // ...
    }
    // ...
    if (x&077 == 0) { // warning: == expression
        // as operand for &
        // ...
    }
}

```

甚至第一个Cfront版本（3.3节）就能产生这些警告。这些都是设计决策的结果，而不是一种事后的高见。

到了很久以后，这些警告中的第一个才被改成了错误：禁止在返回值类型和参数类型处定义新的类型。

2.7 为什么用C

对于带类的C经常提出的一个问题是“为什么用C？为什么你不将它构造在例如Pascal之上？”我的一种答案可以在[Stroustrup, 1986c]中看到：

“很明显，C不是已经设计出来的语言中最清晰的一种，也不是最早使用的，那么为什么还有这么多人使用它呢？

(1) C是很灵活的：可以把C用到几乎所有的应用领域，可以在C中使用几乎所有的程序设计技术。这个语言没有什么内在限制，不排斥在其中写出任何特殊种类的程序。

(2) C是高效的：C的语义是‘低级的’，也就是说，C语言的基本概念直接反映了传统计算机的基本概念。因此，如果想要一个编译系统和/或一个程序员去有效地使用硬件资源，使用C语言相对来说更容易一些。

(3) C是可用的：有了一台计算机，无论是最小的微型机还是最大的超级计算机，情况都一样：那里有可以使用的、具有可接受质量的C编译系统，而且这个编译系统能支持比较完全（可以接受）和标准的C语言和库。也有些可以用的库和工具，这样程序员就不需要从完全空白开始去设计一个新系统。

(4) C是可移植的：C程序通常不能自动地从一种机器（或者一个操作系统）移植到另一种机器上，这种移植通常也不是很容易做的事情。但不管怎么说，这件事通常是可以做的，即使软件的某些重要部分具有内在的机器依赖性，其移植工作困难的程度（从技术的角度或经济的角度）都是可行的。

与这些一级优点相比，那些二级的缺陷，例如C语言古怪的声明语法，某些语言结构缺

乏安全性等等，都变得不那么重要了。设计一个‘更好的C’，就意味着要在不损害C语言优点的情况下，对在写C程序、排错和维护C程序中出现的主要问题做一些矫正。C++保留了所有这些优点以及与C语言的兼容性，代价是断绝了达到完美的念头，付出的还有编译程序和语言的复杂性。无论如何，从空白出发去设计语言也不能保证完美，而C++编译系统具有令人满意的运行时间效率，有更好的错误检查和报告能力，而在代码质量方面也与C编译系统相当。”

与我在带类的C的早期所考虑的东西相比，上面这些说法已经经过很多琢磨。但它确实也反映了我在考虑C时所认识到的最基本的东西，以及我不希望带类的C丧失的东西。Pascal被认为是一种玩具语言 [Kernighan, 1981]。所以，要把类型检查加到C上，与把系统程序设计所必需的特征加到Pascal上相比，应该是更容易也更安全些。那时我有一种对犯错误的恐惧，担心自己作为一个设计者，由于家长式误导或者简单的疏漏，使自己设计出的语言不能用于某些重要领域的实际系统。随后的十年已经很清楚地说明了，选择C语言作为基础使我能站在系统程序设计的主流中，这正是我所期望的。语言复杂性的代价也是相当大的，但是也还能够承受。

那时我考虑过用Modula-2、Ada、Smalltalk、Mesa [Mitchell, 1979]以及Clu代替C作为C++思想的源泉 [Stroustrup, 1984c]，以免忽视了某些灵感。但是只有C、Simula、Algol 68，以及在一种情况下的BCPL在1985年发布的C++里留下了明显的痕迹。Simula提供的是类，Algol 68留下了运算符重载（3.6节）、引用（3.7节）以及在块里任何地方声明变量的能力，而BCPL给出的是//注释形式（3.11.1节）。

在重要方面避免背离C语言有许多理由。我已经看到，要将C作为系统程序设计语言的力量和Simula在组织程序方面的力量结合起来，这本身就是一项很难对付的挑战。如果再从其他来源引进重要特征，那就很容易产生出一种“购物单”式的语言，还会破坏最终语言的完整性。下面的话引自 [Stroustrup, 1986]：

“一个程序设计语言要服务于两种目的：它为程序员提供了一种载体，使他们能描述需要执行的动作；它还提供了一组概念，程序员借助它们思考什么东西是能做的。第一方面的理想是要求一种‘接近机器’的语言，使机器的所有重要方面都能简单而有效地处理，而且是以某种程序员比较容易看清楚的方式。C语言的设计主要就是遵循了这种想法。第二方面的理想是一种‘接近需要解决的问题’的语言，这将使解的概念可以直接而简洁地描述好。加入到C里以创造出C++的那些机制的设计着眼点也就在这个方面。”

同样，与我在设计带类的C的早期所考虑的东西相比，这里的描述也经过了许多琢磨，但其中的普遍性想法那时就已经很清楚了。要脱离C和Simula中那些已知的、已经得到证明的技术，还需要等待带类的C和C++的更多的经验和进一步的试验。我坚定地相信——而且一直相信——语言设计并不是从某个第一原理出发的设计，而是一种需要经验、试验和有效工程折衷的艺术。给语言加入一个主要特征或者概念，也不应该只是信念的一跃，而是一个基于经验的经过深思熟虑的行动，应该考虑怎样才能使它很好地与其他特征构成的框架相配合，以及作为结果的语言应该如何使用等。C++语言在1985年之后的演变，就说明了来自Ada（模板，第15章；异常，第16章；名字空间，第17章），Clu（异常，第16章），以及ML（异常，第16章）的思想的影响。

2.8 语法问题

在C++能够广泛使用之前，我是否在某个时候已经“修正”了C语言语法和语义中最使人讨厌的缺陷？我是否在做这些事的时候没有删除任何有用的特征（在带类的C用户的现实环境里，而不是在某个理想之中）？是否引进了某些不兼容性，而这些东西对于想转到带类的C这边来的C程序员而言是不可接受的？我想是没有。对于某些情况我做过试探，但是在接到受害的用户的抱怨之后，我就退了回来。

2.8.1 C声明的语法

在C语言的语法中，我最不喜欢的就是声明的语法。同时带有前缀和后缀的声明描述符带来了太多的混乱。例如：

```
int *p[10]; /* array of 10 pointers to int, or */
/* pointer to array of 10 ints? */
```

允许省略类型描述符（默认是int）也带来了许多复杂性，例如：

```
/* C style (proposed banned): */
static a; /* implicit: type of 'a' is int */
f(); /* implicit: returns int */

// proposed C with Classes style:
static int a;
int f();
```

要是在这个领域中作修改，用户的否定性反应是非常强烈的。他们把作为C精神的“简练性”推进到如此地步，以至于拒绝使用要求他们写多余的类型描述符的任何一种“法西斯”语言。我从这种修改退了回来，因为没有其他选择。允许隐含的int今天仍然是C++语法中许多最恼人的问题的根源。请注意，这个压力是来自用户，而不是来自管理者或者坐在扶手椅上的语言专家。十年后C++的ANSI/ISO标准化委员会（第6章）决定反对隐含的int。这意味着我们还需要十年左右才有可能摆脱这种东西。有了工具和编译警告的帮助，作为个人的用户现在就可以保护他们自己，防止由于隐含int而造成的混乱了。例如：

```
void f(const T); // const argument of type T, or
// const int argument named T?
// (it's a const argument of type T)
```

在带类的C和C++里采用了在函数括号里写参数类型的函数定义语法，这种方式后来也被ANSI C所采纳：

```
f(a,b) char b; /* K&R C style function definition */
{
    /* ... */
}

int f(int a, char b) // C++ style function definition
{
    // ...
}
```

与此类似，我还考虑了引进线性形式的声明的可能性。C所采用的诡计就是让名字的声明去模仿它们的使用，这就使声明既难读又难写，使人和程序都很容易把声明和表达式弄混。许多人都发现，C语言声明的问题在于声明符*（“指针”）是前缀，而[]（“数组”）和()（“返回函数”）是后缀。这就迫使人们在出现歧义时必须使用括号，例如：

```
/* C style: */  
int* v[10]; /* array of pointers to ints */  
int (*p)[10]; /* pointer to array of ints */
```

与Doug McIlroy, Andrew Koenig, Jonathan Shapiro以及其他人一起，我们考虑引进后缀的“指针”声明符->作为前缀声明符*的替代品：

```
// radical alternative:  
v: [10]->int; // array of pointers to ints  
p: ->[10]int; // pointer to array of ints  
  
// less radical alternative:  
int v[10]->; // array of pointers to ints  
int p->[10]; // pointer to array of ints
```

这种不那么激烈的改变有很大的优点，它允许后缀的->声明符与前缀的*声明符在转变期间同时存在，在这个转变期结束之后，就可以把*声明符和多余的括号从语言里去掉。这种方式的一个显著好处是使括号只用于描述“函数”，这样，造成混乱的机会和语法上的微细差别都被去除了（参见 [Sethi, 1981]）。把所有声明符都变成后缀形式，还能保证声明可以从左向右读，例如：

```
int f(char)->[10]->(double)->;
```

这意味着函数f返回一个指针，该指针指向数组，这是指针的数组，其指针指向返回int指针的函数。请试试写这种直线式的C/C++。不幸的是我把这个想法漏掉了，甚至没有发布过一个实现。与此对应的是人们用typedef逐步构造起复杂的类型：

```
typedef int* DtoI(double); // function taking a double and  
                           // returning a pointer to int  
typedef DtoI* V10[10]; // array of 10 pointers to DtoI  
V10* f(char); // f takes a char and returns  
               // a pointer to V10
```

我最后还是理智地让事情保持其原来的模样，因为任何新语法将进一步（至少是临时性地）增加一个已知为烂泥潭的东西的复杂性。还有，老风格的东西对于喜欢唠叨琐碎事情的教师，或者喜欢嘲弄C的人都是最好的赏赐，对C程序员而言这也不是什么重要问题。在这种情况下我不清楚是否真的做了正确的事。语法的乖僻给我和其他C++实现者、写文档的人以及工具的构造者们带来的苦恼是非常明显的。用户当然可以让自己摆脱这类问题，方式就是只使用C/C++声明的一个小而清楚的子集（7.2节），他们也确实这样做了。

2.8.2 结构标志与类型名

C++引进了一个能给用户带来益处的重要的语法简化，代价是实现语言的人们要多做些工作，还有与C的一些不兼容问题。在C语言里，在结构的名字（结构标志）之前必须出现关键字struct，例如：

```
struct buffer a; /* 'struct' is necessary in C */
```

在带类的C里，这件事让我苦恼了很久，因为这就使得用户定义类型在语法上变成了二等公民。由于我清理语法的其他企图都没有成功，在这里也就犹豫了。后来在Tom Cargill的鼓励下才做了修改——在带类的C向C++演化的时候。在C++里，`struct`，`union`和`class`的名字本身就是类型名，不再需要特定的语法标识符：

```
buffer a; // C++
```

由此造成的与C兼容性的斗争持续了很多年（也参见3.12节）。例如，下面这样的东西在C里是合法的：

```
struct S { int a; };
int S;
void f(struct S x)
{
    x.a = S; // S is an int variable
}
```

它在带类的C和C++里也是合法的。然而在这许多年里我们一直努力想找到一种方式，使它能允许上面这种（几近怪诞，然而也无害的）东西出现在C++里，只是为了兼容性。允许这种例子就意味着必须拒绝：

```
void g(S x) // error: S is an int variable
{
    x.a = S; // S is an int variable
}
```

处理这类特殊问题的实际需求来自这样的事实：某些标准UNIX头文件，特别是`stat.h`，就依赖于让`struct`与某个变量或者函数取同样的名字。这类兼容性问题对于那些语言律师们是重要的，这也正是他们的爱好。不幸的是，在找到一个令人满意——通常又是极端简单——的解决方案之前，这类问题将消耗掉我们无穷无尽的时间和精力。一旦找到了一个解决方案，一个兼容性问题就会变得极端无聊，因为它没有任何内在的智力价值，所具有的不过是实践中的某些重要性。C++对于C多名字空间的解决办法是：一个名字可以指称一个类，同时也可指称一个函数或者一个变量。如果某个名字真的同时指称这两种不同的东西，那么这个名字本身指称的就是那个非类的东西，除非在前面明显地加上关键字`struct`，`union`或者`class`。

因为需要对付倔强的守旧的C语言用户和所谓的C专家，纯正的C/C++兼容性问题是C++开发过程中最困难的也是最受挫折的领域。现在也依然是如此。

2.8.3 语法的重要性

我一直持这种观点：多数人是过分关注语法问题而损害了类型问题。在C++的设计中，最关键的问题总是与类型、歧义性、访问控制等有关，而不是那些语法方面的问题。

这并不是说语法不重要。语法确实非常重要，因为它基本上就是人们可以看到的字面上的东西。一种经过良好选择的语法能大大地帮助程序员学习新概念，避免愚蠢的错误，使他们除了想把东西写正确外，还会更努力地去表述它们。当然，语言的语法设计应该跟着语言

的语义概念走，而不是去绕些弯路。这也就意味着，有关语言的讨论应该集中在它能够表述什么，而不是它怎样去表述。对于什么的回答常常能带来一个有关怎么办的回答，而把注意力集中到语法上，常常就会堕入一种只关系到个人口味的争论。

在有关C兼容性的讨论中有一个很微妙的方面，那就是守旧的C程序员对做事情的老方式总感到很舒服，因此就根本不能容忍为了C本来设计所不能支持的程序设计风格而用的那些不兼容性。与此相对的是，非C程序员通常总会低估C语言语法对于C程序员的价值。

2.9 派生类

派生类是Simula前缀类的C++版本，因此也是Smalltalk中子类概念的兄弟。选用派生 (derived) 和基 (base) 作为名字，是因为我老记不住到底哪个是子 (sub) 哪个是超 (super)，而且我也不是惟一有这种特别问题的人。我也注意到，许多人认为子类比它的超类信息更多是与直觉相矛盾的。在发明术语派生类和基类时，我实际上背离了自己的普遍性原则：只要存在有老术语就不去发明新名字。作为给自己的辩护，我注意到至今还没看到在C++程序员中有弄不清哪个是派生类哪个是基类的问题。这些术语也很容易学习，即使是对那些没有什么数学基础的人。

在提供带类的C的概念时并没有任何形式的运行支持。特别是在这里并不存在Simula（而后C++）的虚函数概念。产生这种情况的原因是我——我想是有很好的理由——怀疑自己是否有能力教会人们如何去使用它，更进一步，是怀疑自己是否有能力使人们相信，在典型的使用中，虚函数在时间和空间方面都与常规函数同样有效。一般来说，有过Simula或者Smalltalk经验的人一直不大相信这一点，直到我们把C++实现的细节解释给他们——即使这样，许多人还是抱着很不合理的怀疑态度。

即使没有虚函数概念，在带类的C里的派生类也很有用，可以用于从老的类构造出新的数据结构，用于将操作与结果类型关联起来，等等。特别是它们使人能够定义各种表类和作业类 [Stroustrup, 1980, 1982b]。

2.9.1 没有虚函数时的多态性

在不存在虚函数的情况下，用户可以使用派生类的对象而把基类当作实现细节。例如，如果有一个向量类，它带有从0开始的下标且没有范围检查 (range checking)：

```
class vector {
    /* ... */
    int get_elem(int i);
};
```

我们可以构造出一个带范围检查的向量，要求元素在某个特定范围内：

```
class vec : vector {
    int hi, lo;
public:
    /* ... */
    new(int lo, int hi);
    get_elem(int i);
};
```

换一种方式，也可以在基类里显式地引进一个类型域，并同时显式地使用类型强制。在用户只看到特殊的派生类而“系统”只看到基类时，应该采用第一种方式。第二种方式用于各种各样的应用类，在这里基类的作用就是为一集派生类实现一个变体记录。

例如，[Stroustrup, 1982b] 给出了下面这段很难看的代码，这里要做的是从一个表里提取一个对象，并基于一个类型域去使用它：

```
class elem { /* properties to be put into a table */ };
class table { /* table data and lookup functions */ };

class cl_name * cl; /* cl_name is derived from elem */
class po_name * po; /* po_name is derived from elem */
class hashed * table; /* hashed is derived from table */

elem * p = table->look("carrot");

if (p) {
    switch (p->type) { /* type field in elem objects */
        case PO_NAME:
            po = (class po_name *) p; /* explicit type conversion */
            ...
            break;
        case CL_NAME:
            cl = (class cl_name *) p; /* explicit type conversion */
            ...
            break;
        default:
            error("unknown type of element");
    }
}
else
    error("carrot not defined");
```

在带类的C和C++里所做的许多努力就是为了保证程序员不必再去写这种代码。

2.9.2 没有模板时的容器类

那时，在我的思想里和我自己的代码中，最重要的就是希望通过基类的组合、显式的类型强制和宏机制（偶然使用）提供出一些通用的容器类。例如，[Stroustrup, 1982b] 展示了可能怎样从一个link的表构造出一个保存某个单一类型对象的表：

```
class wordlink : link
{
    char word[SIZE];
public:
    void clear(void);
    class wordlink * get(void)
        { return (class wordlink *) link.get(); }
    void put(class wordlink * p) { link.put(p); }
};
```

因为通过wordlink而put()进表里的每个link必然是一个wordlink，这样，在从表中用get()取出一个link时，将它强制回到wordlink总是安全的。请注意，这里采用的是私用继承（在描述基类时不写关键字public就默认为私用继承，2.10节）。允许一个wordlink

当作一个普通的link使用有可能损害类型安全性。

宏被用来提供类属类型。下面一段引自 [Stroustrup, 1982b]:

“在引言中，类stack的例子显式地将这个堆栈定义为字符的堆栈。这种东西太特殊了。如果同时又需要一个长整数的堆栈怎么办？如果堆栈类是在某个库里需要的，因此无法知道实际的堆栈元素的类型，那么又该怎么办呢？对于这些情况，类stack的声明和与之关联的函数声明都应该写成某种样子，使元素类型可以在建立堆栈时像参数一样提供，就像有关堆栈大小的参数那样。

在这里没有服务于这种事情的直接的语言支持，而有关效果却可以通过标准的C预处理程序得到，例如：

```
class ELEM_stack {
    ELEM * min, * top, * max;
    void new(int), delete(void);
public:
    void push(ELEM);
    ELEM pop(void);
};
```

这个声明可以放进一个头文件，而对每个具体的ELEM类型只需要做一次宏展开：

```
#define ELEM long
#define ELEM_stack long_stack
#include "stack.h"
#undef ELEM
#undef ELEM_stack

typedef class X;
#define ELEM X
#define ELEM_stack X_stack
#include "stack.h"
#undef ELEM
#undef ELEM_stack

class long_stack ls(1024);
class long_stack ls2(512);
class X_stack xs(512);
```

这当然并不完美，但却是很简单的。”

这是一种最早和最粗糙的技术。对于实际使用而言，它已被证明是太容易引进错误了。所以不久我就定义了几个“标准的”单词粘接 (token-pasting) 宏，并建议对于类属类总用一种基于这些宏的具有特定风格的使用方式 [Stroustrup, 1986, 7.3.5节]。这些技术最终成长为C++语言里的模板机制，以及一些通过模板类和基类去表述实例模板中各种共同性的技术 (第15章)。

2.9.3 对象布局模型

派生类的实现也就是简单地将基类的成员和派生类的成员并列放置。例如，给定了：

```

class A {
    int a;
public:
    /* member functions */
};

class B : public A {
    int b;
public:
    /* member functions */
};

```

类B的一个对象将采用一个结构来表示：

```

struct B { /* generated C code */
    int a;
    int b;
};

```

也就是

int a
 int b

基成员与派生成员之间的名字冲突由编译程序处理，它将在内部给各个成员赋予互不冲突的名字。函数调用被处理得就像根本没有派生时那样。与C语言相比，在这里时间或空间开销也没有增加。

2.9.4 回顾

在带类的C里回避虚函数是合理的吗？是的，没有它们这个语言也很有用。再说，这种东西的缺位也使有关它们的用途、正确使用方法和效率的耗费时日的争论推迟了。缺乏它们也推动人们去开发各种语言机制和技术，这些东西都已经被证明是很有用的，即使有了更强有力的继承机制之后。这些东西也可以作为对某些程序员一概使用继承机制，完全排除了其他程序设计技术的一种制衡力量（见14.2.3节）。特别地，类本来也是为了用于实现实实在在的类型，例如complex或string，界面类也流行起来了。把stack类当作到另一个更一般的dequeue类的界面，这也是没有虚函数的继承的一个例子。

那么，带类的C需要虚函数来服务于它的目标吗？当然，因此这种机制才被加了进来，作为第一个主要扩充，从而造就了C++。

2.10 保护模型

在开始带类的C的工作之前，我一直在操作系统领域工作。来自剑桥CAP计算机和其他类似系统的保护概念——而不是程序设计语言方面的任何工作——激发产生了C++的保护机制。类被作为保护的单位，基本规则是你不能授予自己访问一个类的权力，只有安放在类声明内部的声明（假定是由类的拥有者放置的）可以授予你访问权。按照默认方式，所有的信息都是私用的。

访问权的授予方式就是在类声明的公用部分里声明一个成员，或是把某个特定函数或者

类声明为一个friend。例如：

```
class X {
    /* representation */
public:
    void f();           /* member function with access */
                        /* to representation */
friend void g(); /* global function with access */
                  /* to representation */
};
```

开始时只有类可以作为friend，这样就给该类的所有成员函数赋予了访问的权力。后来发现能把访问权（友关系）授予个别函数也是很方便的。特别地，人们发现能把访问权授予全局函数常常很有用，又见3.6.1节。

友关系被看成是一种机制，类似于一个保护领域将读写的权力授予另一个，这是在类声明中明显写出的，是特指的。因此我确实无法认同那些反复出现的断言，说friend声明“侵犯了封装机制”，我认为这不过是些采用了非C++术语的无知和混乱的结合。

即使是在带类的C的第一个版本中，保护模型也是既适用于成员也适用于基类的。这样，一个类可以以公用方式或者私用方式由另一个类派生。这种对基类的私用/公用区分也化解了延续大约五年的关于实现性继承和界面性继承的争论 [Snyder, 1986][Liskov, 1987]。如果你只想继承一个实现，那么就应该用C++的私用派生。公用派生将使派生类的使用方也能访问由基类所提供的界面。而私用派生方式则把基类留作实现的细节；即使是私用基类的公有成员也是不可访问的，除非经由派生类的界面显式地提供某种方式。

为了能提供“半透明的作用域”，在这里也提供了一种机制，以使私用基类中公用的名字能够以个别的方式进行公用化 [Stroustrup, 1982b]：

```
class vector {
    /* ... */
public:
    /* ... */
    void print(void);
};

class hashed : vector /* vector is private base of hashed */
{
    /* ... */
public:
    vector.print; /* semi-transparent scope */
    /* other vector functions cannot */
    /* be applied to hashed objects */
    /* ... */
};
```

要把一个按一般方式无法访问的名字转为可访问的，其语法形式就是简单写出这个名字。这是具有完美逻辑的、最简单的无歧义语法的一个例子。它又带着某种不必要的模糊性，几乎任何其他语法形式都可能成为对它的改进。这个语法问题现在已经解决了，方法就是引进使用声明（见17.5.2节）。

在 [ARM] 里总结了 C++ 的保护概念：

- 1) 保护是通过编译时的机制提供的，目标是防止发生意外事件，而不是防止欺骗或者有意的侵犯。
- 2) 访问权由类本身授予，而不是单方面的取用。
- 3) 访问权控制是通过名字实行的，并不依赖于被命名事物的种类。
- 4) 保护的单位是类，而不是个别的对象。
- 5) 受控制的是访问权，而不是可见性。

所有这些在 20 世纪 80 年代都是真的，虽然后来有些术语发生了变化。这其中的最后一点可以像下面这样解释：

```
int a;           // global a

class X {
private:
    int a;  // member X::a
};

class XX : public X {
    void f() { a = 1; } // which a?
};
```

如果可见性是受控的，那么，因为 X::a 在这里是不可见的，XX::f() 就会去引用全局的 a。实际上，带类的 C 和 C++ 都认为全局的 a 被不可访问的 X::a 遮蔽了，因此 XX::f() 将产生一个编译错误，因为它企图去访问一个不可访问的变量 X::a。为什么我这样定义呢？这是一种正确的选择吗？我对这一点的回忆是很模糊的，留下的记录也无法说明问题。有一点我确实记得，那时在一个讨论中给出了上面这个例子，所采纳的规则能保证 f() 对 a 的引用将总是引用同一个 a，不管对 X::a 所声明的访问权如何。如果让 public/private 控制可见性（而不是访问权），那么一个从公用到私用的修改就会不动声色地改变程序的意义，从一种合法解释（访问 X::a）变成了另一种（访问全局的 a）。现在我不再认为这个论据具有排他性（如果我过去这样认为的话），但是所做出的这种决策已经被证明是很有用的，因为它允许程序员在排错时加上或者去掉 public 或 private 描述，而又不会在暗地里改变程序的意义。我确实怀疑 C++ 定义的这个方面是否为一个真正设计决策的结果。它也可能是在采用预处理器技术实现带类的 C 时引出的一个简单的自然后果，而后在用更合适的编译技术实现 C++ 时（3.3 节）又没再做仔细的审查。

C++ 保护机制在另一个方面也受到操作系统的影响，这里的规则具有容忍欺骗的趋向。我假定任何有能力的程序员都可以欺骗所有不是由硬件强制规定的规则，因此，在这里不需要去防范欺骗 [ARM]：

“C++ 的访问控制是为了防止意外事件而不是防止欺骗。任何程序设计语言，只要它支持对原始存储器的访问，就会使数据处于一种开放的状态，使所有有意按照某种违反数据项原本类型规则所描述的方式去触动它的企图都能够实行。”

保护系统的职责就是保证所有违反类型系统的操作都只能显式地进行，并且尽量减少这类操作的必要性。

操作系统中读/写保护的概念还融入了 C++ 的 const 概念（3.8 节）。

多年来也出现了许多建议，希望能提供对小于整个类的单元的保护，例如：

```
grant X::f(int) access to Y::a, Y::b, and Y::g(char);
```

我一直在抗拒这类建议。我的基本想法是这种小粒度控制不会增加任何保护：一个类里的任何成员函数都能修改这里的任一个数据成员。所以，如果一个函数将控制权授予另一个函数成员，就可以间接地修改每个成员。我那时就认为（现在依然），由更多显式控制得到的利益抵不上由它带来的在描述、实现和使用方面的复杂性。

2.11 运行时的保证

上面描述的访问控制机制都是为了防止未经授权的访问。第二类保证则通过“特殊的成员函数”提供，例如建构函数，这些函数是由编译程序识别和调用的。基本想法就是想让程序员能够建立起一种保证，有时也被称为不变式，以便使其他成员函数都能够依赖这些东西（参见16.10节）。

2.11.1 建构函数与析构函数

那时我常用这样的方式解释有关概念，有一个“新建函数”（建构函数，constructor），它建立起其他成员函数进行操作的环境基础；另有一个“删除函数”（析构函数，distructor）来销毁这个环境，并释放它以前获得的所有资源。例如：

```
class monitor : object {
    /* ... */
public:
    new()    { /* create the monitor's lock */ }
    delete() { /* release and delete lock */ }
    /* ... */
};
```

另见3.9节和13.2.4节。

建构函数的概念从何而来？我怀疑这个词是自己发明的。我过去就熟悉Simula的类对象初始化机制。但无论如何我是把类声明主要看成一个界面的定义，因此就希望能避免把代码放在这里面。由于带类的C与C语言一样有三种存储类型，几乎必然需要有某种由编译程序识别的初始化函数形式（2.11.2节）。观察发现，允许定义多个建构函数很有价值，因此这也就成了C++重载机制的一个重要应用方面（3.6节）。

2.11.2 存储分配和建构函数

与在C语言里一样，对象可以用三种方式分配：在堆栈上（在自动存储区），在固定的地址（静态存储区），以及在自由空间里（在堆里，或说动态空间里）。在所有这些情况下都必须调用建构函数，以建立起这个对象。在C语言里，在自由空间分配一个对象时只牵涉到调用一个分配函数。例如：

```
monitor* p = (monitor*)malloc(sizeof(monitor));
```

对于带类的C，这显然是不够的，因为这样做无法保证一定会调用建构函数。因此我引进了一个运算符，以便保证分配和初始化都能够完成：

```
monitor* p = new monitor;
```

该运算符被称为new，因为这也是Simula里对应运算符的名字。new将调用某种分配函数以获得存储，而后调用一个建构函数去初始化这些存储。这种组合操作常常被称为实例化，或者简单地称为对象创建，它从原始的存储区建立起一个对象。

运算符new的记法规定也很重要（3.9节）。但是，将分配和初始化组合为一个操作，没有一种显式的错误报告机制也带来了一些实际问题。很少需要在建构函数中处理和报告错误，当然，异常机制（16.5节）的引进为这个问题提供了一种具有普遍意义的解决办法。

为了尽量减少重新编译，如果对一个带建构函数的类去使用new运算符，Cfront就将其实现为简单地调用对应的建构函数，让这个建构函数去完成分配和初始化工作。这意味着，如果在某个翻译单元里对类x的所有对象都是用new分配的，而且没有调用x里的任何在线函数，那么如果x的大小或者表示形式改变了，这个翻译单元也不必重新编译。翻译单元是ANSI C的术语，意指预处理之后的源程序文件，也就是在一次分别编译中提供给编译程序的那些信息。我发现，组织好自己的模拟程序，以便尽可能地减少重新编译是一件非常有用的事情。但是，尽量减少重新编译的问题并没有得到带类的C和C++团体的普遍重视，这种情况一直延续了很长时间（13.2节）。

运算符delete()被引进来，作为new的对应物，就像释放函数free()是与malloc()对应的东西一样（3.9节，10.7节）。

2.11.3 调用函数和返回函数

也许有些奇怪，在带类的C的最初实现里有一种特征是C++所没有提供的，但它也是人们经常需要的。人们可能需要定义一个函数，在各成员函数（除建构函数之外）的每个调用之时这个函数都应该被隐含调用；也可能需要定义另一个函数，在各成员函数（除析构函数之外）的每次调用返回时都需要调用它。它们被称作调用函数（call）和返回函数（return）。它们被用在原始的作业库管程类里提供同步 [Stroustrup, 1980b]。

```
class monitor : object {
    /* ... */
    call() { /* grab lock */ }
    return() { /* release lock */ }
    /* ... */
};
```

这些代码在意图上类似于CLOS[⊖]里的:before和:after方法。调用和返回函数后来从语言里去掉了，因为除了我之外没人用它们；也因为我感觉到要使人都相信call()和return()非常有用可能会完全失败。在1987年Mike Tiemann提出了另一种解决方案，称为“包装（Wrapper）” [Tiemann, 1987]，但是在Estes Park举行的USENIX实现者工作会议上，这种思想被认为存在着太多的问题，不能被接收到C++里。

2.12 次要特征

两个次要特征，赋值的重载和默认参数也被引进了带类的C。它们是C++语言中重载机制

[⊖] CLOS，Common Lisp Object System，是Lisp语言的一种面向对象的扩充。——译者注

(3.6节) 的前辈。

2.12.1 赋值的重载

有一个情况很快就被注意到了：具有非平凡表示的类（如string或vector）无法成功地进行复制，因为C语言赋值的语义（按位复制）对于这些类型都是不正确的。这种默认的复制语义所建立的实际上是一种共享表示，而不是真正的副本。我对这个情况的反应就是允许程序员自己描述复制的意义 [Stroustrup, 1980]。

“很不幸，这种标准的结构赋值方式并不总是理想的。典型情况是，一个类的对象本身只不过是一棵信息树的根，简单地复制这个树根而不关心任何分支往往不是我们所希望的。与此类似，简单地去复写一个类的对象也很可能造成混乱。

允许为一个类的对象改变赋值的意义，实际上就提供了一种处理这类问题的方法。要做这件事，只要声明一个称为operator=的类成员函数。例如：

```
class x {
public:
    int a;
    class y * p;
    void operator = (class x *);
};

void x.operator = (class x * from)
{
    a = from->a;
    delete p;
    p = from->p;
    from->p = 0;
}
```

这是为类x的成员定义了一个采用分解方式的读写[⊖]动作，与标准语义的隐含复制操作是完全不同的。”

在 [Stroustrup, 1982]的那个版本里用的是另一个例子，其中检查了this==from，以便能正确地处理自我复制问题。很明显，我学到这种技术也很不容易。

有了定义之后，这个赋值运算符就会被用于实现所有显式写出的和隐含的复制操作。原来对初始化的处理首先是初置为一个默认值（通过一个不带参数的new函数，建构函数），而后再做赋值。这种方式被认定是低效率的，这就引出了后来C++里面的复制建构函数（11.4.1节）。

2.12.2 默认参数

用户定义的赋值运算符蕴涵着对默认构造函数的大量使用，这自然就引出了默认参数的问题 [Stroustrup, 1980]：

“默认参数表是很晚才加到类机制里的，加上它是为了抑制一类情况：为了将类对象作为

⊖ 原书中为read，不够清楚，因为赋值中既牵涉到读，也牵涉到写。——译者注

函数的参数，或者为处理实际是其他类的成员的类对象，或者是为了处理基类的参数，这些情况下都可能出现大量完全相同的‘标准参数表’。在这些情况中提供参数表已经被证明是很讨厌的事情，为避免这种麻烦就需要另外引进一种‘特征’，它能用于把class对象的声明弄得更简洁些，更像struct声明。”

这里是一个例子：

“完全可以为一个new函数声明一个默认的参数表，当需要建立这个被声明的类的对象，而又没有提供参数的时候，就使用这个表：

```
class char_stack
{
    void new(int=512);
    ...
};
```

这就使声明：

```
class char_stack s3;
```

成为合法的，在初始化s3时调用的是s3.new(512)。”

给出了一般性的函数重载（3.6节，第11章）之后，默认参数在逻辑上已经变成多余的，至多不过是一种次要的记法规则罢了。当然，在一般性的重载出现在C++中之前，默认参数已经在带类的C里使用了许多年。

2.13 考虑过，但是没有提供的特征

早期还考虑过许多特征，其中一些后来出现在C++里，有的仍然在讨论中。这其中包括虚函数、static成员、模板和多重继承等。

“所有这些推广都有它们的用处，但是语言的每种‘特征’都需要占用时间，并要耗费精力去设计、实现、写文档、还有学习……。基类的概念是一种工程折衷，就像C的类概念^Θ [Stroustrup, 1982b]。”

我现在还希望那时就能明确地提出需要更多的经验，只有依靠它才能抵抗形式主义，并使这种讲究实际的方式比较完整。

在1985之前的一段时间里也考虑过自动废料收集的可能性，但后来还是相信，对一个已经被用在实时处理和硬核心系统（例如设备驱动程序）的语言而言，这种特征是不合适的。在那段日子里废料收集还没有今天这样的复杂，而与今天的系统相比，一般计算机的处理能力和存储容量也是非常低的。我自己使用Simula的个人经验以及其他关于基于废料收集的系统的报告都使我相信，对于我和我的同事所要写的这类应用系统而言，废料收集是无法承受的东西。如果带类的C（甚至是C++）被定义成需要自动废料收集的语言，它一定会更优雅些，但也会是一个死胎。

还考虑过直接支持并发的问题，但是我也拒绝了，因为我更喜欢基于库的方式（2.1节）。

^Θ 原书如此。这实际上应该是指带类的C里的类概念。——译者注。

2.14 工作环境

带类的C作为一个研究项目，是我在贝尔实验室的计算科学研究中心设计和实现的。这个中心提供了一—并仍然提供着—做这种工作的绝好环境。当我加入到那里时，告诉我的是“做点什么有趣的事情”，提供了合适的计算机资源，被鼓励去与有兴趣并且有能力的人们交谈。还给了我一年时间，在此之后在才正式展示自己的工作、接受评估。

在这里有一种文化倾向，反对需要许多人参加的“伟大的项目”；也反对“伟大的计划”，例如要求其他人去实现的未经测试的论文式设计；还反对在设计者和实现者之间的阶级划分。如果你喜欢那些东西，在贝尔实验室和其他许多地方你都可以沉溺于这种爱好之中。但是，计算科学研究中心则总是要求你—如果你不是做理论的话—个人实现出某些融入了你自己的想法的东西来，并能设法找到一些用户，他们能从你构造出的东西中获益。这个环境对于做这类工作是非常合适的。在这个实验室里有一大批人，他们都有许许多多的思想和问题需要去挑战，愿意去测试任何构造出来的东西。正因为此，我才会在 [Stroustrup, 1986] 中写到：

“从来都没有一个论文式的C++设计：设计、文档和实现是同时进行的。很自然，C++的前端也是用C++写的。从来就没有一个‘C++项目’，或者一个‘C++设计委员会’。从一开始，C++就一直在发展中，现在还在继续发展，通过作者与他的朋友和同事的讨论，去克服用户遇到的问题。”

只是到了C++已经成为一个建立起来的语言之后，某些常见的组织结构才出现了。即使在这之后，我还是正式地负责参考手册，并且掌握着将什么东西放进去的最后决定权。这种状况一直持续到1990年，那时有关事项才移交给了ANSI的C++委员会。作为标准化委员会中扩展小组的主席，我一直对进入C++的每个特征负直接责任。而在另一方面，在工作开始的几个月之后我就再也没有设计的自由了，例如不能为了把某些东西搞得更美妙些而去更改设计，或者随意对当时的语言做一些修改。每个语言特征都要求一个实现，使之变成现实的东西。而任何改动或扩充都需要得到带类的C，以及后来C++的关键用户们的同意，通常也是笃信。

因为不存在用户群能够扩大的保证，这个语言及其实现要想生存，就必须以最好的方式服务于用户的需要，这样才能抵御已有语言的有组织的拉力，以及市场上有关最新语言的宣传。特别地，即使是要引进某个不大的不兼容性，也必须要求给用户带来某种大得多的利益。因此，即使是在语言开发早期也没有经常引进较大的不兼容性。对用户而言，几乎每个不兼容性都可能被认为是很重要的，因此我一直尽可能少地引进不兼容性，只有在从带类的C到C++的转移时，才有意地打破了许多程序。

虽然缺乏正式的组织结构，缺乏大规模的资金、人力、“牢牢拴住的”用户和市场的支持，但是，我有那些非正式的帮助和我在计算科学研究中心同事们的见识，有中心管理方面提供的非技术的需求和开发组织上的保护，这些足以弥补前面的那些缺憾。如果没有中心成员们的见识和与社会喧嚣的隔离，C++的设计肯定会向各种时尚和特殊利益妥协，它的实现也会陷入官僚主义的泥潭之中。同样重要的是，贝尔实验室提供了一个环境，在那里不需要为了个人升迁而隐藏起自己的思想。相反，讨论在过去能够也确实是在自由地进行，现在依然如

此。这就使人们能够从其他人的思想和观点中获益。不幸的是，计算科学研究中心的情况并不很典型，即使是在贝尔实验室的内部。

带类的C就是在与计算科学研究中心的人们的讨论中成长起来的，其早期用户分布在实验室里的各个地方。带类的C以及后来的C++的大部分都是在别的什么人的黑板上做出来的，其他东西是在我的黑板上。大部分想法都被否定了，因为过于精巧，使用上太受限制，太难实现，太难教会人们用到实际项目里，在空间或者时间上的效率太低，与C语言太不兼容，或者简单地就是太离奇。很少的一些想法能够通过这种过滤，一律要经过与至少两个人的讨论，而后我再去实现。典型情况是，想法的成熟要通过在实现和测试两方面的努力，而后是我和另外几个人的使用。这样得到的结果版本在一个更大的听众群中试验，通常在更成熟一点之后才被放进“正式的”带类的C里由我发布。一般来说，在这期间还要写出一个教材。写教材也被看成是一种最基本的设计工具，因为如果某个特性很难有简单的解释，支持它就会是一个沉重的负担。这一点也与我个人密切相关，因为在早期的那些年里我就是那个支撑组织。

在工作的早期，Sandy Fraser（那时我所在部门的负责人）的影响很大。例如，我认为就是他促使我脱离Simula的类定义风格（其中包含着完整的函数定义），转而采纳了另一种风格，典型情况下是把函数定义放在其他地方，以强调类声明作为一个界面的角色。带类的C的许多设计是为了构造模拟器，以便能用在Sandy Fraser有关网络设计的工作中。带类的C最早的实际应用就是这种网络模拟器。Sudhir Agrawal是另一个早期用户，他也通过自己在网络模拟方面的工作影响了带类的C的开发。Jonathan Shopiro基于他有关“数据流数据库机器”的模拟工作，对带类的C的设计和实现提出了许多反馈意见。

如果要做关于程序设计语言更一般问题的讨论，而不是寻找应用或者确定哪些问题需要解决，我就转去找Dennis Ritchie、Steve Johnson，特别是Doug McIlroy。Doug对于C和C++开发的影响无论怎么估计也不会过分。我不记得有任何一个关于C++的重要设计决策没有与Doug讨论过。当然，我们的意见并不总是一致，但是我对做一个与Doug观点相反的决策总是非常不情愿。他有把握正确观点的特质，还有看起来是无穷无尽的经验和耐性。

由于带类的C和C++的主要设计是在黑板上做出来的，有关的思考倾向于集中在对“典范问题”的求解上，将一个小例子看作是刻画了一大类问题的基本特性。这样，对这种小问题的一种好的解决方案，将会为写解决与此类似的实际问题的程序提供很大帮助。通过我在自己的文章、书籍和讲演中用这些东西作为实例，许多这样的问题已经进入了C++的文献和民间传说之中。对于带类的C，考虑过的最关键的例子是task类，它是支持完成具有Simula风格的模拟的一个作业库的基础。其他关键的类是queue、list和histogram。queue和list类都基于由Simula借来的一种思想，提供一个link类，让用户基于这个类派生自己的类。

这种途径也隐含着一个危险：这样做有可能构造出一个语言和工具，对小的经过精心选择的例子，它们能够提供很优雅的解，但却无法扩张，无法去构造完整的系统或者大的程序。这个问题被简单地解决了，事实是，带类的C（以及后来的C++）在其早期就必须完成自己的实现。这保证了带类的C不会演变为某种貌似优雅但是却毫无用处的东西。

作为一个密切联系用户的个人工作，它也给了我一种自由，只许诺自己实际能够提供的东西，而不必把我的许诺扩大到某种程度，以便使它看起来能对某个组织具有经济上的意义，

以便能得到为开发、支持以及市场化一个“产品”所需要的各种重要资源。像所有语言一样，C++也需要为了生存而在其儿童时代就开始参加工作，它在成熟时带着一条实践和现实主义的绶带，也带着累累伤痕。基于作业库的对于网络、电路版布局、芯片、网络规程等等的模拟，这些就是在早期年代里我的面包和黄油。

第3章 C++的诞生

任何纽带的连接力都比不上遗传链。

——Stephen Jay Gould

从带类的C到C++——Cfront, C++的初始实现——虚函数和面向对象的程序设计——运算符重载和引用——常量——存储管理——类型检查——C++与C的关系——动态初始化——声明的语法——C++的描述和评价

3.1 从带类的C到C++

到1982年,有一件事我已经很清楚了:带类的C只是一个“中等的成功”,它也能保持这种状态直到它死亡。我把中等成功定义为某一种东西,它很有用,能够支付其本身和开发者的开销,但是又没有足够的吸引力和有用性,不足以负担起一个支持和开发组织。这样,如果继续与带类的C和它的C预处理器实现共事下去,也就是宣判我去无穷无尽地支持带类的C。我当时只看到两条摆脱这种两难境地的路:

(1) 停止支持带类的C,使那些用户必须转到其他地方去(于是我得到了自由,可以去做别的事情了)。

(2) 基于我在带类的C上的经验,开发一种新的更好的语言,使它能够服务于一个足够大的用户群,能够支付一个支持和开发组织(这也就使我得到自由,可以去做别的事情)。那时我估计5000个工业用户是必需的最小值。

通过市场方式(广告宣传)增加用户人数这样的第三条途径从来也没有出现在我的考虑之中。实际发生的事情是C++(后来对这个新语言的命名)的爆炸性成长,这使我一直如此繁忙,以至到今天还没能从这里脱身去做别的什么重要事情。

我认为,带类的C的成功是其设计目标的自然推理:带类的C确实能有助于把一大类程序组织得比C语言好得多,而且又没有损失运行效率。它也没有因为要求剧烈的文化转变,而使各种不希望发生重要转变的组织认为它不可行。也有一些制约它成功的因素,一方面是它所提供的超越C的新功能集合太有限,还有就是在实现带类的C时使用的是预处理器技术。对于准备向重要的方向投资,以便能获得相当收获的人而言,带类的C不能给他们足够的支持:带类的C是向着正确方向迈出的重要一步,但它只不过是一小步。作为这种分析的结果,我开始设计一个带类的C的后继语言,做一些清理和扩充,并要用传统的编译技术去实现它。

这样产生的语言一开始还是叫带类的C。为满足管理部门提出的一个有礼貌的要求,它被重新命名为C84。这样命名的原因是人们已经开始将带类的C称为“新的C”,而后直接称它为C。这种简化又使C语言被称作“简单的C”、“直接的C”以及“老的C”。特别是这最后一个名字被认为是侮辱性的。所以,无论是作为一般的礼貌,还是希望避免混乱,都促使我去寻找一个新的名字。

名字C84也只用了几个月，一方面是因为这个名字太丑陋，过于制度化，也是因为如果人们丢掉84还是会引起混乱。同时Larry Rosler（负责C语言标准化的X3J11 ANSI委员会的编辑）也请求我另外找个名字。他解释说：“标准化语言常用的称呼方式就是用它们的名字，而后是标准的年份。如果有一个超集（C84，原来的带类的C，而后是C++）带有比它的子集（C，可能是C85，而后是ANSI C）更小的编号，那是很使人为难的，也是迷惑人的。”这看起来绝对是合理的。虽然Larry后来对C标准的发布日期有了更乐观的态度，我还是开始在带类的C的用户团体里征求对新名字的想法。

我采用了C++是因为它很短，有一种很好的解释，而且不是那种“形容词+C”的形式。在C语言里++（根据上下文）可以读作“下一个”、“后继者”或者“增加”，虽然它总是被读作“加加”。名字C++与其竞争者++C一直是玩笑和双关语的丰富源泉，在这个名字被确定之前，这些几乎都是大家所熟知并非常欣赏的。C++的名字是Rick Mascitti建议的。它的第一次使用是在1983年12月，那时这个名字被编辑进[Stroustrup, 1984]和[Stroustrup, 1984c]的最后拷贝中。

C++里的C有一段很长的历史。当然它是Dennis Ritchie所设计的语言的名字。C的直接前驱是BCPL类语言的一个解释性后代：由Ken Thompson设计的B。BCPL是剑桥大学的Martin Richard在访问位于另一个剑桥的MIT期间设计的，BCPL反过来又是Basic CPL，这其中的CPL是另一个更大的（在那时）而且是很优雅的程序设计语言的名字，这个语言由剑桥大学和伦敦大学合作开发。在伦敦的人们参加这个项目之前，“C”表示剑桥。后来“C”正式的说法是Combined（结合）。而按照非正式的说法，“C”表示Christopher，因为Christopher Strachey是CPL背后的主要动力。

3.2 目标

在从1982到1984的这段时间里，C++的目标逐渐变得更野心勃勃也更加确定了。我已经开始把C++看成C语言之外的另一种语言，库和工具也作为工作领域开始出现了。因为这些情况，因为在贝尔实验室内部的工具开发者们已经开始表现出对C++的兴趣，也因为我已经在着手一个完整的新实现Cfront，它将成为C++编译系统的前端，这时我必须回答下面这几个问题：

- 1) 用户将是哪些人？
- 2) 他们将使用哪些种类的系统？
- 3) 我怎样才能避免提供工具的工作？
- 4) 对于1)、2)、3)的回答将会怎样影响语言的定义？

我对1)，“用户将是哪些人？”的回答是：首先是我在贝尔实验室的朋友和我将使用它；而后在AT&T内部更广泛的使用，以便能提供更多的经验；再往后一些大学会取走有关的思想和工具，最后AT&T和其他人将能够销售那些成形的工具集合。而到了某个时候，我将要做出的初始的试验性实现将逐渐淡出，让位于由AT&T或别的什么地方开发的更具产业强度的实现。

这些在实践上和经济上看都是很有道理的。初始的（Cfront）实现将是缺少工具的、可移植的和廉价的，因为这是我、我的同事和许多大学的用户们所需要的，也是能负担得起的。再往后应该有更好的工具和更专业化的环境的丰富来源。这些更好工具的目标主要是工业用

户，因此就不必也是廉价的，因此就能够承担起为大规模地使用这个语言所需要的支持组织。这就是我对问题3), “我怎样才能避免去提供工具？”的回答。简而言之，这种策略也行通了，但是所有细节实际上是以当时无法预料的方式发生的。

为得到对2), “他们将使用哪些种类的系统？”的回答，我简单地环顾四周，观察带类的C的用户们实际使用的是哪些系统。他们使用各种系统中的所有东西，系统可能非常小，以至他们无法运行一个为主机或者超级计算机而做的编译器。他们使用的操作系统比我听说过的还多。由此我总结出需要最强的可移植性和交叉编译的能力，也不能对运行结果代码的机器的大小和速度有任何假定。为了构造一个编译系统，无论如何我也需要对人们用于开发程序的系统类型做一些假设。我假定有一个MIPS加上一兆字节可以用。我认为这个假设是有点危险的，因为大部分我最希望的用户在那时还在共用着PDP11机器，或者其他能力相对更低的系统以及/或者分时系统。

我没有预料到PC革命，但是为了超过为Cfront所设定的性能目标，我碰巧做出了一个编译系统，它（勉强）能在IBM PC/AT上运行。这就提供了一个现成的证据，说明C++能够成为PC上的高效语言，同时也激励商品软件的实现者去超过它。

作为对4), “这些回答会怎样影响语言的定义？”的回答，我的结论是不能带有要求特别复杂的编译系统或运行支持的特征，必须能使用原来可用的连接程序，而且要求产生的代码一开始就应该高效（与C相比较）。

3.3 Cfront

C84语言的Cfront编译前端是我在1982年春天到1983年夏天这段时间里设计和实现的。计算机科学研究中心之外的第一个用户是Jim Coplien，他在1983年7月接到自己的拷贝。Jim当时在伊利诺依州Naperville的贝尔实验室的一个组工作了一段时间，用带类的C做试验性的开关网络。

在同一时期我设计了C84，我撰写的参考手册在1984年1月1日出版 [Stroustrup, 1984]，设计了complex复数库，并和Leonie Rose [Rose, 1984] 一起实现了这个库，和Jonathan Shapiro一起设计并实现了string库，维护和移植带类的C实现，为带类的C用户提供支持，并帮助他们转变为C84的用户。这一年半真是非常忙。

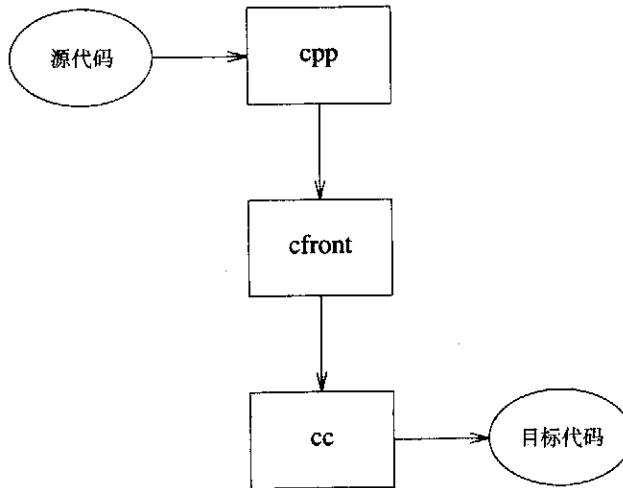
Cfront是一个传统的编译系统前端，它执行对语言语法和语义的完全检查，构造起一个对应于输入的内部表示，分析并重新安排这个表示，最后产生出适合某个代码生成程序的输出结果。这里用的内部表示是一个图，对每个作用域有一个符号表。一般策略是从源文件里一次读入一个全局声明，直到分析完一个完整的全局声明时才产生输出。

在实践中，这样做就意味着编译系统需要有足够的内存去保存所有全局名字和类型的表示，再加上对应一个函数的完整的图。几年后我对Cfront做了定量测试，发现在DEC VAX上它的存储使用量稳定在大约600 KB的水平上，几乎与提供给它的实际程序无关。正是这个事实使我在1986年很容易地把Cfront移植到了PC/AT上。在1985年发布Release 1.0时，Cfront大约包含12 000行C++代码。

Cfront的组织方式是很传统的，除了它有可能使用多个符号表（未必是一个符号表）之外。Cfront最初是用带类的C写的（还能用什么呢？），不久就转到用C84描述，所以很早的能工作的C++编译系统本身就是用C++写的。甚至在Cfront的第一个版本里就已经大量地使用了类和

派生类。它没有用虚函数，因为在项目刚开始时还没有这种机制。

Cfront（只）是一个编译前端，只有它是无法做实际程序设计的。还需要有一个驱动程序，先让源程序通过C语言预处理器Cpp，而后让Cpp的输出通过Cfront，最后让Cfront的输出经过一个C编译器：



还有，这个驱动程序必须保证（动态的）初始化一定能够完成。到Cfront 3.0时，这个驱动程序也变得更复杂了，因为其中实现了自动的模板实例化（15.2节）[McCluskey, 1992]。

3.3.1 生成C

Cfront最不平常的方面——在它那个时候——就是它生成C语言的代码。这也带来了无穷无尽的混乱。让Cfront生成C代码，是因为我希望这个初始实现具有最强的可移植性，而我认为C是环顾四周可以看到的最可移植的汇编语言。我很容易从Cfront生成某种内部的后端格式的东西，或者生成汇编代码，但这都不是我的用户所需要的东西。没有任何一种汇编系统或者编译后端能够为我的用户团体中多于四分之一的人服务，也不存在一种方式使我能生成（例如）所需要的六种不同后端形式，以便能满足这个团体中90%的人。为了应付这样的需求，我的结论是用C作为一种公共的输入形式，供给各种代码生成程序使用，这是惟一的合理选择。这种把编译系统做成一个C代码生成系统的方法后来变得很流行，许多语言，如Ada、Eiffel、Module-3、Lisp、Smalltalk都用这种方法做过实现。这样我得到了高度的可移植性，编译时付出的额外代价也是合适的。有关代价的来源包括：

- 1) Cfront把作为中间表示的C代码写出来所需要的时间。
- 2) C编译系统为读入中间的C代码所需要的时间。
- 3) C编译系统分析中间的C代码所“浪费”的时间。
- 4) 控制整个过程所用的时间。

这个代价主要依赖于读写中间的C表示所花费的时间，而这又主要依赖于系统的磁盘读写策略。在这些年中，我在多种系统上测量过这个开销的情况，发现它大致是编译“必需时间”的25%到100%的样子。我也看到过一些不采用C语言中间形式的C++编译器比用Cfront再加上C编译器还要慢。

请注意，这里的C编译器仅仅是用作一个代码生成器。从C语言编译器产生的任何错误信息，所反映的或者是C编译器本身错误，或者就是Cfront的错误，绝不会是C++源代码的。所有语法或语义错误原则上都由C++的前端Cfront捕捉。在这方面，C++及其Cfront实现不同于基于预处理器的语言，如Ratfor [Kernighan, 1976] 和 Objective C [Cox, 1986]。

我强调这些是因为人们对Cfront是什么的混乱认识有很长的一段历史。因为它产生C代码就说它是一个预处理器。那些在C团体里（或者其他地方）的人们以此作为证明的根据，说Cfront是个相当简单的程序——就像一个宏预处理器。人们还据此（错误地）“推论”说从C++到C的逐行翻译是可能的，使用Cfront时不可能在C++语言的层次上做符号排错工作，由Cfront生成的代码必然不如“真正的编译系统”所生成的代码，C++不是一个“真正的语言”，如此等等。很自然，我早就对这些毫无根据的论断感到厌倦了——特别是当它们被用来批判C++语言时。有几个C++编译系统现在还是用Cfront加上一个本地的代码生成器，而不是再经过一个C前端。对于用户而言，这样做的明显差异只是编译速度快了一些。

我绝对不喜欢大部分形式的预处理器和宏。C++的一个目标就是使C语言的预处理器变成多余的（4.4节，第18章），因为我认为这种操作具有产生错误的倾向。Cfront的基本目标就是让C++具有合理的语义，而这种语义是不能用那时的C语言编译器实现的：这种编译器对类型和作用域知道的很不够，无法完成C++所需要的那一类解析。C++的设计非常强烈地依赖于传统的编译技术，而不是依赖运行时支持或者程序员对表达式的细节解析（就像你在使用没有重载的语言里所面临的一样）。因此，C++不可能用任何传统的预处理技术进行翻译。我当时曾考虑过这种为语言提供语义的方式以及翻译器技术，后来拒绝了它们。Cfront的直接前辈Cpre就是一个相当传统的预处理器，它完全不懂得C的任何语法、作用域和类型规则，这成为语言定义和实际使用中许多问题的根源。我后来就决定不在修订后的语言和新的实现里再看到这些问题。C++和Cfront是一起设计的，语言定义和编译技术确实互相影响，但却不同于那些按简单方式思维的人所设想的情况。

3.3.2 分析C++

在1982年我刚开始计划Cfront的时候，我想用一个递归下降分析器，因为过去有写这种东西和维护它们的经验；也因为我喜欢这种分析程序，它们具有良好的错误信息生成能力；还因为我在考虑哪些东西必须做到分析器内部时，喜欢这种可以使用通用程序设计语言完整能力的想法。但是作为一个认真的年轻计算机科学家，我还是想到去问专家。Al Aho和Steve Johnson当时在计算机科学研究中心。他们，特别是Steve，使我确信手工地去写分析程序已经大大落后于时代了，简直就是浪费时间，可能导致不很系统化，以至不可靠，出现许多错误。正确的方式是使用一个LALR(1)分析程序生成器，因此我使用了Al和Steve的YACC [Aho, 1986]。

对大多数项目而言这应该是正确的选择。对于几乎所有的从空白起步的试验性语言项目而言，这应该是正确的选择。对于大多数人而言，这也应该是个正确的选择。回过头看，对于我和C++而言这个选择却是一个很严重的错误。C++并不是一个新的试验性语言，它几乎是一个与C语言兼容的C的超集。而那时还没有人能写出一个C的LALR(1)语法。ANSI C所用的LALR(1)语法是大约一年半后由Tom Pennello构造出来的——太晚了，我和C++都无法从中受益。即使是Steve Johnson的PCC（这是那个时候最前卫的C编译器），也在一些对写C++分

析器的人们真正是麻烦事的地方耍了小骗局。例如，PCC不能正确处理多余的括号，因此 `int (x);` 就不能被接受为一个 `x` 的声明。更坏的情况是，看起来有的人特别擅长于某些分析策略，而另一些人则在另外一些策略上更好些。我的个人爱好是自上而下分析，历年中多次以某种构造性的形式使用它，而这种东西很难装进一个YACC语法里。至今Cfront还是用着一个YACC分析器，它以许多基于递归下降技术的词法分析技巧作为补充。从另一方面看，为C++写一个高效的、具有合理美感的递归下降分析器是完全可能的，一些时髦的C++编译器采用了递归下降技术。

3.3.3 连接问题

前面已经提到过，我已经决定在传统连接程序的限制之内活动。但是在这里也有一个限制，我虽然能够忍受，但是它却实在太愚蠢了。我如果有足够的耐心，现在与它斗争的机会已经来了。大部分传统连接程序都把外部名字能够使用的字符数限制到非常少，只允许8个字符的限制是很常见的。在K&R C里只保证可以用具有一种大小写形式的6字符的外部名字，ANSI C也接受了这个限制。成员函数的名字需要包括它所在的类的名字，而重载函数的类型也必须以这种或那种方式反映到连接过程中（参见11.3.1节），我确实没有其他办法。

考虑：

```
void task::schedule() { /* ... */ } // 4+8 characters
void hashed::print() { /* ... */ } // 6+5 characters
complex sqrt(complex); // 4 character plus 'complex'
double sqrt(double); // 4 character plus 'double'
```

要把这种名字用至多6个大写字符表示就必须做某种形式的压缩，这将使工具的构造进一步复杂化，也可能要涉及到某种形式的散列，这样就需要用一个基本的“程序数据库”去解决散列溢出问题。前一种方式很令人生厌，而后一种方式也会引起一个严重问题，因为在传统的C/Fortran连接模型中并没有“程序数据库”的概念。

因此我就开始（在1982年）鼓动在连接程序里使用更长的名字。我不知道自己的活动是否有实际效果，但是在这段时间里，大部分连接程序确实给出了更大一些的字符数限制，而这正是我所需要的。Cfront以某种编码的方式实现类型安全的连接，对这种方式，32个字符对于方便的使用还显得太少了，甚至256有时也会觉得紧张（参见11.3.2节）。在此期间，对于老式的连接程序临时性地用了一个对长标识符的散列编码系统，但这样做总不能令人完全满意。

3.3.4 Cfront发布

第一个带类的C实现和第一个C++实现都是在很早的版本时就跨出了贝尔实验室，一些大学系科的人们直接向我要。基于这种途径，几十个教育单位的人们开始使用带类的C。例如斯坦福大学（1981年12月，第一个Cpre发布），加州伯克莱大学，在麦迪逊的威斯康星大学，加州理工，在查伯海的北卡罗莱纳大学，麻省理工，悉尼大学，卡内基-梅隆大学，在厄尔巴纳-琛佩恩的伊利诺伊大学，哥本哈根大学，罗斯福实验室（牛津），IRCAM，INRIA等等。在C++的设计和实现之后，又继续进行这种对个别教育单位的发布活动。例如加州伯克莱大学（1984年8月，第一个Cfront发布），华盛顿大学（圣路易斯），奥斯汀的得克萨斯大学，哥

本哈根大学，新南威尔士大学，等等。此外，学生们也显示出他们避免纸面工作的创造性。以至到了后来，处理这种个别的发行变成了我的一个负担，也成了大学中等待C++的人们烦恼的原因。因此我的部门负责人Brian Kernighan、AT&T生产主管Dave Kallman和我达成了一个想法，就是做出一种Cfront的更具普遍意义的发布。其想法就是尽量避免商业问题，如确定价格，写合同，提供支持，做广告，取得学生证明文件等等。采用的方法是给大学的人们按照发行所用的磁带价格提供Cfront和几个库。这被称为Release E，其中的E是指教育。第一批磁带是在1985年1月发给有关机构的，例如罗斯福实验室（牛津）。

Release E打开了我的眼界。事实上，Release E是翻了车。我曾经希望对C++的兴趣在大学里能掀起狂风大浪。但是C++用户的增长还是按照正常的曲线（7.1节），我们看到的并不是新用户的洪水，而是从大学教授来的抱怨的洪水，因为C++不是商业上可以使用的东西。我一次次接触并听到“是的，我想使用C++，我也知道可以免费地得到Cfront。但不幸的是我无法使用它，因为我需要某种可以用在我的咨询里面的东西，需要某种我的学生可以用在工业中的东西。”为推动学习的纯粹学术工作做得太多了。Steve Johnson（后来是负责C++的部门负责人）、Dave Kallman和我又回到图板，回到为商业发布的Release 1.0计划。但无论如何，将“几乎免费”的C++实现（带着源程序和库）提供给教育机构的政策（原来是用Release E）的方式还一直保持着。

C++语言的版本一直用Cfront的发布编号命名。Release 1.0是《C++程序设计语言》(*The C++ Programming Language*) [Stroustrup, 1986] 所定义的语言。Release 1.1(1986年6月)和1.2(1987年2月)基本上只是修正了一些错误的更新版本，但也加上了指向成员的指针和保护成员(13.9节)。

1989年7月的Release 2.0是一个大清理，还引进了多重继承(12.1节)。这个版本被广泛认为是一个重要的进步，无论是在功能上还是在质量上。Release 2.1(1990年4月)基本上是一个修正错误的版本，它也使C++进入了《带标注的C++参考手册》(*The Annotated C++ Reference Manual*) [ARM] 定义的轨道(5.3节)。

Release 3.0(1991年9月)加上了在ARM里描述的模板机制(第15章)。3.0的一个能支持ARM所描述的异常处理机制(第16章)的变形是Hewlett-Packard公司的产品 [Cameron, 1992]，在1992年后期开始发行。

我写了Cfront的第一个版本(1.0、1.1和1.2)并维护它们。Steve Dewhurst在1985年的Release 1.0之前和我一起做了几个月。Laura Eaves在Release 1.0、1.1、2.1和3.0的Cfront分析程序上做了许多工作。我还做了Release 1.2和2.0的很大一部分程序设计，从Release 1.2开始，Stan Lippman在Cfront上面花掉了他的大部分时间。Laura Eaves, Stan Lippman, George Logothetis, Judy Ward和Nancy Wilkinson做了Release 2.1和3.0的大部分工作。1.2、2.0、2.1和3.0是由Barbara Moo管理的。Andrew Koenig组织了Cfront 2.0的测试。从Object Design Inc.来的Sam Haradhvala在1989年做了模板的初始实现，Lippman为1991年的Release 3.0扩充了这个实现。Cfront里的异常处理机制是Hewlett-Packard公司在1992年做的。除了这些人所做的代码最后进入了Cfront的主要版本之外，还有许多人从它出发做了一些局部的C++编译系统。在这些年里，许多公司，包括Apple, Centerline(以前的Saber), Comeau Computing, Glockenspiel, ParcPlace, Sun, Hewlett-Packard和其他公司也都发布了一些产品，其中包括了Cfront的本地化修改版本。

3.4 语言特征

在带类的C的基础上引进了一些新特征，从而形成了C++，它们是：

- 1) 虚函数 (3.5节)
- 2) 函数名和运算符重载 (3.6节)
- 3) 引用 (3.7节)
- 4) 常量 (3.8节)
- 5) 用户可控制的自由空间存储区控制 (3.9节)
- 6) 改进的类型检查 (3.10节)

此外，调用和返回函数 (2.11节) 被去掉了，因为它们没什么用处。还修改了许多细节，以产生出一个更清晰的语言。

3.5 虚函数

虚函数是C++里最显著的新特征，它也必然是对人们用这个语言进行程序设计的风格影响最大的东西。虚函数的思想是从Simula借来的，以一种适当的方式出现在这里，希望能比较容易做出简单而有效的实现。需要虚函数的理由在 [Stroustrup, 1986] 和 [Stroustrup, 1986b] 里有阐述，为了强调虚函数在C++程序设计中的核心作用，我在这里要从 [Stroustrup, 1986] 引证一些细节：

“一个抽象数据类型定义了一种黑盒子。一旦定义好之后，它就不会实际地与程序其他部分产生交互作用了。没有其他办法能为某些新的用途而调整它，除非是修改它的定义。这可能带来很严重的灵活性问题。考虑一个为在图形系统里使用而定义的shape类型。假定目前情况下系统必须支持圆、三角形和矩形，再假定你已经有了一些类：

```
class point{ /* ... */ };
class color{ /* ... */ };
```

你可能把shape类定义成下面的样子：

```
enum kind { circle, triangle, square };

class shape {
    point center;
    color col;
    kind k;
    // representation of shape
public:
    point where()      { return center; }
    void move(point to) { center = to; draw(); }
    void draw();
    void rotate(int);
    // more operations
};
```

这里“表示种类的域”k是必需的，以便draw()和rotate()一类函数能确定它们当时正在处理的形状 (shape) 的种类 (在Pascal语言里我们可以用一个带标志k的变体记录)。函数draw()的定义可能像下面这样：

```

void shape::draw()
{
    switch (k) {
    case circle:
        // draw a circle
        break;
    case triangle:
        // draw a triangle
        break;
    case square:
        // draw a square
        break;
    }
}

```

这真是一个烂泥潭。像draw()这样的函数必须“理解”现存的所有形状。这样，每当有一种新形状被加入系统时，所有这类函数的代码都必须扩充。如果你定义了一个新形状，处理形状的每个操作都必须检查并（做可能需要的）修改。如果你无法接触所有这些操作的源代码，就无法将一个新的形状加入到系统中。加入一个新形状牵涉到“触及”与形状有关的各种重要操作的代码，这需要高超的技艺，也很可能将潜在的错误引进处理其他形状的代码里。为特定的形状选择表示方式也将受到很严格的束缚，因为它们的表示（至少其中的一部分）必须能放进由通用类型shape的定义所规定的固定框架里面。

实际问题是，在这里没有区分任意形状的普遍性质（例如，一个形状有一种颜色，它可以被画出来，等等）和与某种特定形状有关的属性（例如，圆是一种有圆心的形状，应该由一个画圆的函数来画，等等）。面向对象的程序设计就是要表达这种差异并从中获益。如果在某种语言里存在能够表述和利用这种差异的结构，它就能支持面向对象的程序设计。其他语言则不行。

Simula的继承机制提供了一种解决方案，我把它移植到了C++里。首先需要描述一个类，用它定义所有形状的普遍性质：

```

class shape {
    point center;
    color col;
    // ...
public:
    point where() { return center; }
    void move(point to) { center = to; draw(); }
    virtual void draw();
    virtual void rotate(int);
    // ...
};

```

那些只能在这里给出调用界面定义，而又必须针对每个特定类型专门定义实现的函数，在这里都需要标明virtual（按照Simula和C++的说法，“可以在将来由这个类派生的类重新定义”）。有了这个定义，我们就可以写出对各种形状进行操作的通用函数了：

```

void rotate_all(shape** v, int size, int angle)
    // rotate all members of vector "v"
    // of size "size" "angle" degrees
{

```

```

    for (int i = 0; i < size; i++) v[i]->rotate(angle);
}

```

如果要定义一个特定类型，我们必须说明它是一个形状，而后再描述它的特殊性质（包括那些虚函数）。

```

class circle : public shape {
    int radius;
public:
    void draw() { /* ... */ };
    void rotate(int) {} // yes, the null function
};

```

在C++里，类circle被说成是由类shape派生的。而shape被称为circle的基类。另一种方式是分别把shape和circle称作子类和超类。”

关于虚函数和面向对象程序设计的进一步讨论，请见13.2节、12.3.1节、13.7节、13.8节和14.2.3节。

我不记得在那时人们对虚函数有多大兴趣，可能是我没把其中涉及的概念解释清楚，我从附近的人们那里得到的主要反应是忽视和怀疑。有一种常见的观点，说虚函数不过是某种蹩脚的函数指针，因此完全是多余的。更糟的是有时人们争辩说，设计良好的程序根本不需要虚函数所提供的那些可扩充性和开放性，所以，只要通过正确分析就总能弄清楚应该调用哪个非虚函数。随后有关的意见又有了发展，说虚函数只不过是一种低效的形式。我当然不同意这些，还是把虚函数加了进来。

我有意地没有在C++里提供一种显式获取对象类型的机制：

“没有把Simula 67的INSPECT语句引进C++中是故意的。这样做的原因就是为了鼓励通过虚函数的使用去实现模块化 [Stroustrup, 1986]。”

Simula的INSPECT语句是一种基于系统提供的类型域的开关语句。我已经看到过太多由于使用这种机制而产生的错误，因此确定应该尽可能依靠静态的类型检查和C++的虚函数。一种运行时的类型获取机制最后还是被加进了C++（14.2节），我希望这种机制的形式将使它不像INSPECT语句在Simula中那样吸引人。

3.5.1 对象布局模型

这里最关键的实现思想，就是把在一个类里定义的一集虚函数定义为一个指向函数的指针数组。这样，对虚函数的调用简单地也就是通过该数组一个间接函数调用。对每个有虚函数的类都存在一个这样的数组，一般称为虚函数表或者vtbl。这些类的每个对象都包含一个隐式指针，一般称为vprt，指向该对象的类的虚函数表。有了：

```

class A {
    int a;
public:
    virtual void f();
    virtual void g(int);
    virtual void h(double);
};

```

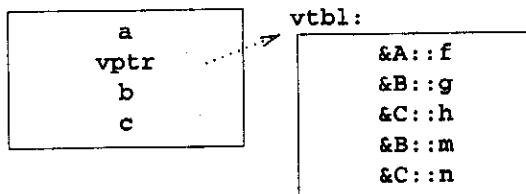
```

class B : public A {
public:
    int b;
    void g(int); // overrides A::g()
    virtual void m(B*);
};

class C : public B {
public:
    int c;
    void h(double); // overrides A::h()
    virtual void n(C*);
};

```

类C的一个对象看起来大概是这样：



对虚函数的调用被编译系统翻译成一个间接调用。例如：

```

void f(C* p)
{
    p->g(2);
}

```

将变成某种类似下面的东西：

```
(* (p->vptr[1])) (p, 2); /* generated code */
```

这并不是唯一可能的实现方式。这种方式的优点是简单和运行的高效；它的问题是如果你更改了一个类的虚函数集合，所有使用它的代码都必须重新编译。

至此，对象模型在某种意义上已经成为真实的东西了，因为对象已经不再是它的类的数据成员的一个简单汇集。带有虚函数的C++类的对象已经成为与C语言里简单的struct完全不同的另一种东西了。然而，为什么在那个时候我没有做出选择，把结构和类看成不同的概念呢？

我的意图是只有一个概念：一个统一的布局规则集合，一个统一的检索规则集合，一个统一的解析规则集合，如此等等。我们或许也可以在两组规则之下生活，但是用一个概念能将有关特征与更简单的实现平滑地集成到一起。我当时就确信，如果struct对用户意味着“C和兼容性”，而class则是“C++和高级特征”，那么整个团体就会分裂成两个阵营，并很快就停止互相交流。在设计类的时候，能根据自己的需要选用尽可能多或尽可能少的语言特征，这种思想对我一直是很重要的。只有采用一个概念才能支持我的想法，得到一种从“传统的C程序设计”，通过数据抽象达到面向对象程序设计的平稳而渐进的过渡。只有采用一个概念，才能支持“你只为你所用的东西付出代价”的理想。

回过头看，我认为对于C++成为一种成功的实际工具，这些概念是非常重要的。在这些年里，几乎每个人都有过某种代价沉重的想法去实现为“专门为类所使用的服务”，而把低代价

和低级特征留给struct。我认为，让struct和class始终作为同一个概念，实际上也将我们拯救了出来，没有去把类实现为一种支持高代价、多样化、具有与目前很不相同的特征的东西。换句话说，正是“struct就是class”的概念使C++没有随波逐流，变成另一种带着一个没有什么联系的低级子集的更高级的语言。有些人可能曾经希望这种事情发生。

3.5.2 覆盖和虚函数匹配

虚函数只能被派生类里的函数覆盖，该函数应该具有同样的名字，同样的参数和返回类型。这样就避免了任何形式的运行时参数类型检查，也不需要在运行时保存大量的类型信息。例如：

```
class Base {
public:
    virtual void f();
    virtual void g(int);
};

class Derived : public Base {
public:
    void f();      // overrides Base::f()
    void g(char); // doesn't override Base::g()
};
```

这也为不谨慎者打开了一个陷阱：非虚的Derived::g()实际上与Base::g()无关，但是却覆盖了它。如果你用的编译系统不能对这类问题提供警告，这个情况可能就真正成为一个问题了。当然，编译程序是很容易检查出这种问题的，对那些能够提供警告的实现，这个问题也就不存在了。Cfront 1.0不能提供警告，因此带来了一些麻烦。Cfront 2.0和更高的版本都能做出这种警告。

原来的规则对覆盖函数是要求有准确的类型匹配，后来又对返回类型有所放松，参见13.7节。

3.5.3 基成员的遮蔽

派生类里的名字将遮蔽基类中具有相同名字的任何对象或函数。这不是一个好的设计决定，这件事成为多年来许多争论的主题。这个规则最先是由带类的C引进的，我把它看作是普通作用域规则的必然推论。在有关这个问题的争论中，我一直认为与此相反的意见——将派生类和基类的名字合并到同一个作用域里——至少也会带来许多问题。特别是有可能错误地对子对象调用改变状态的函数：

```
class X {
    int x;
public:
    virtual void copy(X* p) { x = p->x; }
};

class XX: public X {
    int xx;
public:
    virtual void copy(XX* p) { xx = p->xx; X::copy(p); }
```

```

};

void f(X a, XX b)
{
    a.copy(&b); // ok: copy X part of b
    b.copy(&a); // error: copy(X*) is hidden by copy(XX*)
}

```

如果合并了基类和派生类的作用域，那么第二个copy操作也就允许做了，而这将导致b的状态被部分更改的问题。大多数情况下这将导致对XX对象的操作出现非常奇怪的行为。我确实看到过人们被这种方式套住，他们使用的是GNU的C++编译系统（7.1.4节），该编译系统允许这种重载。

如果copy()是虚函数，我们可以认为XX::copy()覆盖了X::copy()，而这时就需要做运行时的检查才能捕捉到b.copy(&a)的问题，程序员也必须按防御的方式编程，以便能在运行中捕捉到这种错误（13.7.1节）。当时已经理解了这种情况，但我害怕还有什么自己尚未理解的问题，因此就选择了现在这种最严格、最简单和最有效的规则。

回过头再看，我也怀疑在2.0中引进的覆盖规则（11.2.2节）或许就能处理这类情况。考虑调用b.copy(&a)，变量b在类型上正好与XX::copy()的隐含参数完全匹配，但需要经过一个标准转换才能与X::copy()匹配。变量a则是另一种情况，它正好与X::copy()的显式参数类型匹配，但要经过一个标准转换才能与XX::copy()匹配。这样看，如果允许这种覆盖，那么这个调用就是错误的，因为它有歧义性。

参见17.5.2节，在那里给出了一种显式地要求基类函数和派生类函数重载的方法。

3.6 重载

一些人早就要求有重载运算符的能力。运算符重载“看起来很舒服”，而我根据自己在Algol 68上的经验，也知道如何让这种想法能够工作。但是我对于把这种重载引进C++还是很犹豫的，因为：

- 1) 人们普遍认为重载很难实现，可能导致编译系统膨胀到可怕的规模。
- 2) 人们普遍认为重载很难进行教学，也很难精确地定义。这样，手册和教材就可能膨胀到可怕的规模。
- 3) 人们普遍认为使用运算符重载方式写出的代码的效率比较低。
- 4) 人们普遍认为重载会使代码阅读比较困难。

如果3)和4)是对的，那么C++不提供重载反而会更好一些。如果1)和2)是对的，我将没有资源去提供重载机制。

但是，上面的这些推测都是错误的，重载确实能为C++用户解决某些实际问题。也确实存在这样的用户，他们希望在C++里有复数、矩阵、APL语言里那样的向量。存在这样的用户，他们希望有检查范围的数组、多维数组以及字符串。至少出现过两个不同的应用，在其中人们希望能重载逻辑运算符，如|（或）、&（与）和^（不相交或）[⊖]。我这样看这件事情：随着C++用户人数的规模增加和工作的多样化，上面这种列举的表将变得更长更大。我对问题4）（重载会使代码难以阅读）的回答来自我的几个朋友，我非常敬重他们的观点，他们的

[⊖] 我们都知道这些是按位运算运算符，应该是将它们看成二进制位上的逻辑运算符。——译者注

经验是以十年来计算的。他们说，如果有了重载，他们的代码会变得更清晰。那么，如果人们能用重载写出难读的代码又怎么样呢？在任何语言里都可以写出很难读的代码。一种特征能够怎样被用好，要比它可能怎样被用错更重要得多。

下面一步，我也使自己确信了重载并不是固有的低效率 [Strousrup, 1984b] [ARM, 12.1c]。关于重载机制的细节大部分是在我的黑板和 Stu Feldman, Doug McIlroy 和 Jonathan Shopiro 的黑板上做出来的。

这样，在做出了一个对于3)（用重载写出的代码是低效的）的回答之后，我还需要关注1) 和2)，编译系统和语言的复杂性问题。我首先研究了带重载运算符的类（例如 `complex` 和 `string`）的使用情况，用起来相当容易，不会给程序员带来很大的负担。而后我写出了手册中有关的几节，证明所增加的复杂性不是什么严重问题，手册增加了不到一页半（整个手册是42页）。最后，我用了两个小时就做出了第一个实现，在 `Cfront` 里只增加了18行代码。这样我才觉得已经证明了过去对定义和实现的复杂性方面的恐惧是过分夸大了。但无论如何，在第11章里还是要说明重载确实存在着一些问题。

当然，实际上所有这些问题并不是严格按照这个顺序处理的，这里不过是想强调工作如何逐渐从使用的问题转移到实现的问题。重载机制的细节在 [Stroustrup, 1984b] 里描述，使用这种机制的一些类的例子写在 [Rose, 1984] 和 [Shopiro, 1985] 里。

重新审视这些情况，我认为运算符重载是C++语言里最主要的一种财富。除了在各种数值应用中算术运算符（+、*、+=、*=，等等）重载的明显使用方式外，[] 下标、() 函数应用、= 赋值常常被用于控制访问，而 << 和 >> 变成了标准的I/O运算符（8.3.1节）。

3.6.1 基本重载

这里是一个展示有关的基本技术的例子：

```
class complex {
    double re, im;
public:
    complex(double);
    complex(double,double);

    friend complex operator+(complex,complex);
    friend complex operator*(complex,complex);
    // ...
};
```

这将使简单的复数表达式可以被解析为函数调用：

```
void f(complex z1, complex z2)
{
    complex z3 = z1+z2; // operator+(z1,z2)
}
```

赋值和初始化也不需要显式地定义。按照默认方式，它们被定义为按逐个成员的方式复制，参见11.4.1节。

在重载机制的设计中，我依靠转换去减少所需要的重载函数的个数。例如：

```
void g(complex z1, complex z2, double d)
{
```

```

complex z3 = z1+z2; // operator+(z1,z2)
complex z4 = z1+d; // operator+(z1,complex(d))
complex z5 = d+z2; // operator+(complex(d),z2)
}

```

这就是说，我依靠从double到complex的隐式转换，只用一个复数的加函数就支持了各种“混合模式算术”。也可以再引进额外的函数，以改进效率或者数值精度等。

原则上说，即使没有隐式类型转换我们也完全可以做：或者是明显地要求做转换，或者是提供完全的复数加运算集合：

```

class complex {
public:
    friend complex operator+(complex,complex);
    friend complex operator+(complex,double);
    friend complex operator+(double,complex);
    // ...
};

```

没有隐式转换我们能否做得更好呢？没有这种东西，语言可以更简单一些；隐式转换也必定会被过度地使用；一个涉及转换函数的调用，在典型的情况下必定比一个准确匹配的调用效率稍微低一些。

让我们考虑四个基本算术运算。定义有关double和complex的完整的混合运算符集合需要12个算术函数，与之相对的是，采用隐含转换只需要4个函数再加上一个转换函数。在那些所涉及的操作和类型数目更多的地方，一边是通过使用转换而得到的函数数量的线性增长，另一边是由于需要所有组合而出现的平方式的爆炸，差别是极其显著的。我看到过这样的例子，因为不能安全地定义转换，在那里提供了完整的运算符集合，结果是超过100个定义运算符的函数。我认为，对于极特殊的情况这还是可以接受的，但这种东西不应该成为一种标准的实践。

当然我也认识到，并不是所有的建构函数都定义了有意义、不会使人感到意外的隐式转换。例如，vector类型通常都有这样一个建构函数，它有一个整数参数指明元素的数目。这个函数就有一种很不幸的副作用：v=7将构造出一个7个元素的vector并用它给v赋值。不过我并不认为这个问题非常急迫。C++标准化委员会的一些成员，特别是Nathan Myers，所建议的一个解决办法并不是必要的。到1995年这个问题被解决了，采用的方法是允许给建构函数声明加上前缀explicit。一个声明为explicit的建构函数只能用于显式的对象构造，不能用于隐式转换。例如，把vector的建构函数声明为explicit vector(int);，就能够使v=7产生一个错误，而显式的v=vector(7)则仍然是普通的构造和给v的赋值。

3.6.2 成员和友元

请注意，上面定义的operator+是一个全局函数（一个友元函数），而没有采用成员函数。这样做是为了保证+的运算对象能被对称地处理。如果用的是成员函数，我们的解析将具有下面的样子：

```

void f(complex z1, complex z2, double d)
{
    complex z3 = z1+z2; // z1.operator+(z2);
}

```

```

    complex z4 = z1+d; // z1.operator+(complex(d))
    complex z5 = d+z2; // d.operator+(z2)
}

```

这就要求我们给出将complex加到内部类型double上的定义。这样不仅需要有更多的函数，也将要求在许多地方修改代码（在类complex的定义里和内部类型double的定义里）。这肯定不是我们所希望的。我考虑了允许为内部类型定义额外运算的问题，但后来还是拒绝了这种想法，因为我不想改变规则：任何类型——无论是内部的还是用户定义的——都不能在其定义完成之后再增加额外的操作。在这里还有另一个原因：C内部类型之间的转换定义已经够肮脏了，绝不能再向里面添乱。而通过成员函数提供混合模式算术的方式，从本质上讲，比我们所采纳的全局函数加转换函数的方式还要肮脏许多。

采用全局函数就使我们能定义这样的运算符，它们的参数具有逻辑的对称性。与此相对应的是把运算定义为成员，这样能够保证在调用时对第一个（最左的）运算对象不出现转换。这使我们能模拟那种要求以左值作为左运算对象的运算符，例如赋值运算符：

```

class String {
    // ...
public:
    String(const char*);
    String& operator=(const String&);
    String& operator+=(const String&); // add to end
    // ...
};

void f(String& s1, String& s2)
{
    s1 = s2;
    s1 = "asdf"; // fine: s1.operator=(String("asdf"));
    "asdf" = s2; // error: String assigned to char*
}

```

Andrew Koenig后来注意到，如+=一类的赋值运算符是更基本的，它们也比其常规算术的兄弟+等等的效率更高些。最好是先把+=、*=一类的赋值运算符函数定义为成员函数，而后再把+、*一类的常规运算符定义为全局函数：

```

String& String::operator+=(const String& s)
{
    // add s onto the end of *this

    return *this;
}

String operator+(const String& s1, const String& s2)
{
    String sum = s1;
    sum+=s2;
    return sum;
}

```

请注意，在这里并不需要友元关系，二元运算符的定义也非常简单、具有统一的风格。在实现对+=的调用时不需要临时变量，局部变量sum则完全是用户必须考虑的一种临时变量管理

方式。剩下的东西都能由编译系统简单而有效地处理了(3.6.4节)。

我原来的想法是允许每个运算符都可以是成员的或者是全局的。特别是我发现将简单的访问函数定义为成员函数，而后让用户把他们自己的运算符实现为全局函数是很方便的。我的讨论对+或-这样的运算符都有效，但是对运算符=本身就会遇到问题。因此，在Release 2.0里要求运算符=必须是成员。这是一个不兼容的修改，它打破了某些程序，所以这个决定绝不能小看。这里的问题是，除非=是成员，否则在一个程序里就可能根据在源代码里的位置得到对=的两种不同解释。例如：

```

class X {
    // no operator=
};

void f(X a, X b)
{
    a = b; // predefined meaning of =
}

void operator=(X&, X); // disallowed by 2.0

void g(X a, X b)
{
    a = b; // user-defined meaning of =
}

```

这可能造成很大的混乱，特别是当两个这样的赋值出现在分别编译的不同源文件里的时候。因为对类没有预先定义的+=的意义，因此就不会出现类似问题。

在另一方面，我甚至在C++的初始设计里就把[]、()和->限制为必须是成员。这应该是一种无害的限制，它能清除掉一些出现隐晦错误的可能性，而这些运算符通常总是要修改它们左运算对象的内部状态。当然，这也可能是一种不必要的谨小慎微。

3.6.3 运算符函数

既然已经决定支持隐式转换以及由它们所支持的混合模式运算，我就需要有一种方法去描述这类转换。只有一个参数的建构函数就是这样的一种机制。给出了

```

class complex {
    // ...
    complex(double); // converts a double to a complex
    // ...
};

```

我们就可以显式地或者隐式地把一个double转换到complex。当然，只有类的设计者能够定义这个类的转换。在定义一个新类时，要使它能融入某个已有的框架也是很常见的事情。例如，在C语言库里有数十个以字符串为参数的函数，也就是说其参数类型都是char*。当Jonathan Shopiro第一次写功能完全的String类时，他发现，或者是需要去重写C库里所有以字符串为参数的函数：

```

int strlen(const char*); // original C function
int strlen(const String&); // new C++ function

```

或者是要在C++里提供一个从String到char*的转换运算符。

我理所当然地在C++中增加了转换运算符的概念：

```
class String {
    // ...
    operator const char*();
    // ...
};

int strlen(const char*); // original C function

void f(String& s)
{
    // ...
    strlen(s); // strlen(s.operator const char*())
    // ...
}
```

在实际使用中，隐式转换的使用有时确实被证明是有些诡异的，但是去提供混合模式的完整运算符集合也一点不漂亮。我当然希望能有一种更好的解决办法，但在我知道的所有方法里，隐式转换就是瘤子里挑出的将军。

3.6.4 效率和重载

与人们经常表现的朴素迷信不同，在把操作表示为函数调用或是将操作表示为运算符之间并没有什么根本性的差别。与重载有关的效率问题主要是在线化和避免多余的临时变量。

为了使自己能够确认这些情况，我首先注意到，例如，由 $a+b$ 或 $v[i]$ 生成的代码，与由例如 $add(a, b)$ 或 $v.elem(i)$ 这样的函数调用生成的代码完全一样。

随后我又观察到，程序员可以通过在线化去保证简单操作不会带来函数调用的开销（无论是在时间上还是在空间上的）。最后我又观察到，为了有效地支持对大的对象做这种风格的程序设计，引用调用机制是必需的（在3.7节有更多的讨论）。剩下的问题就是如何避免在例如 $a=b+c$ 中出现多余的临时变量。它生成

```
assign(add(b, c), t); assign(t, a);
```

这当然比不上

```
add_and_assign(b, c, a);
```

编译系统对内部类型可以生成这样的代码，程序员也可以直接写出这种代码。最后，我在[Stroustrup, 1984b]里展示了如何生成以下语句。

```
add_and_initialize(b, c, t); assign(t, a);
```

这里还剩下一个“多余的”复制操作，只有在 $+$ 和 $=$ 操作都不实际地依赖于被赋的值时（没有别名时）。这个复制才能够去掉。有关这种优化的最容易找到的参考材料见[ARM]。直到Cfront的Release 3.0这种优化才能用。我相信，第一个采用这种技术的C++实现是Zortech的编译系统。1990年一次C++标准会议后，在西雅图的天针[⊖]顶上吃水果奶油冰淇淋时我把这个问题解释给Walter Bright，随后他很容易地就实现了这种优化。

⊖ Space Needle，指西雅图的电视塔。——译者注

我认为这种稍微差一点的优化模式也是可以接受的，因为有更明确的运算符如`+=`，借助于它们可以对最常见的情况进行手工优化。此外，在做初始化时可以假定并不存在别名问题。从Algol 68借来了一个观念是声明可以出现在任何需要它们的地方（而不仅仅是在某些分程序的开始处），这样就可以推动一种“仅做初始化的”或“一次赋值的”程序设计风格，这种方式具有其内在的有效性，与反复给变量赋值的传统方式相比也更不容易出错。例如，可以写

```
complex compute(complex z, int i)
{
    if ( /* ... */ ) {
        // ...
    }
    complex t = f(z, i);
    // ...
    z += t;
    // ...
    return t;
}
```

而不是更罗嗦也更低效的：

```
complex compute(complex z, int i)
{
    complex t;
    if ( /* ... */ ) {
        // ...
    }
    t = f(z, i);
    // ...
    z = z + t;
    // ...
    return t;
}
```

参见11.6.3节，那里提出了另一种提高运行效率的想法，也是要努力去掉临时变量。

3.6.5 变化和新运算符

我认为最重要的是应该把重载操作作为一种扩展语言的机制，而不是改变语言的机制。也就是说，应该能定义在用户定义类型（类）上工作的运算符，而不是修改内部类型原有运算符的意义。进一步说，我也不想允许程序员引进新的运算符。我害怕像密码似的记法，也害怕采用非常复杂的语言分析策略（像Algol 68所需要的那样）。正因为这些，我认为自己的限制是合理的。参见11.6.1节和11.6.3节。

3.7 引用

引入引用机制主要也是为了支持运算符的重载。Doug McIlroy还记得，有一次我向他解释某个预示了目前运算符重载模式的问题。他用的术语引用挑起了我的思绪，我嘟囔了一声谢谢就离开了他的办公室。当我第二天再出现时就带着已经基本完成的目前模式。Doug使我想起了Algol 68。

C语言对所有函数参数都采用值传递，如果用值传递对象的效率太低或者不合适，用户就

可以传递一个指针。在有了运算符重载后，采用这种策略就不行了，因为在这种情况下写法上的方便是最本质的东西，对于大的对象也不能指望使用者去加进所需要的取地址运算符。例如：

```
a = b - c;
```

是可以接受的（也就是说，是方便的）写法，但

```
a = &b - &c;
```

就不行。当然，`&b - &c`在C语言里已经有了预定的意义，我也不想去改变它。

在初始化之后，被一个引用所引用的东西就不能再改变了。也就是说，一旦一个C++引用被初始化后，你就无法使它再去引用另一个对象；或者说，它不能被重新约束。我过去曾经被Algol 68的引用刺痛过，在那里写`r1=r2`是通过`r1`给被引用的对象赋值，或者给`r1`赋一个新的引用值（重新约束`r1`），具体是什么要看`r2`的类型。我希望在C++里避免这种问题。

如果希望在C++里做更复杂的指针操作，那么就可以用指针。因为在C++里同时有指针和引用，因此就不需要区分对引用本身的操作和对被引用对象的操作（像在Simula里那样），也不需要采用Algol 68里的那种推理机制。

我那时也犯了一个严重错误，那就是允许用非左值的东西去初始化非`const`引用。例如：

```
void incr(int& rr) { rr++; }

void g()
{
    double ss = 1;
    incr(ss);    // note: double passed, int expected
                  // (fixed: error in Release 2.0)
}
```

由于类型不同，`int&`不能直接引用传递来的`double`，因此就需要产生一个临时变量，在其中保存`ss`值所对应的`int`值。这样`incr()`修改的将是这个临时变量，修改结果不会反映到做调用的那个函数里去。

允许用非左值的东西做引用的初始化，原本是希望能把按值传递和按引用传递之间的差别变成由被调用函数描述上的细节问题，使调用函数不需要去关心它。对`const`引用做这种考虑是合适的，而对非`const`引用就不行了。C++的Release 2.0在这里做了些修改，就是反映了这个情况。

允许对`const`引用使用需要转换类型的非左值和左值进行初始化，这一点也非常重要。特别是这样就使我们可以用常量去调用Fortran的函数：

```
extern "Fortran" float sqrt(const float&);

void f()
{
    sqrt(2); // call by reference
}
```

除了引用参数这种最明显的使用方式之外，我们认为能够以引用作为返回类型也是非常重要的。这将使我们能为字符串类定义很简单的下标运算符：

```
class String {
```

```

// ...
char& operator[](int index); // subscript operator
                           // return a reference
};

void f(String& s, int i)
{
    char c1 = s[i]; // assign operator[]'s result
    s[i] = c1;       // assign to operator[]'s result
    // ...
}

```

返回的是一个到String内部表示的引用，这里实际上假定用户对所做的事情是负责任的。在许多情况下这种假设也是合理的。

左值和右值

通过重载operator[]()使它返回一个引用，在这里，不允许写operator[]()的人为通过下标对指定元素的读/写操作提供不同的语义。例如：

```
s1[i] = s2[j];
```

我们不能对被写的字符串(s1)做一种操作，而对被读的字符串(s2)做另一种操作。Jonathan Shopiro和我都认为，允许为读访问和写访问分别提供语义也是一种很基本的东西，当时考虑的是访问带共享表示的字符串和数据库的问题。在这两种情况中，读都是简单而廉价的操作，而写则是具有潜在的高代价的复杂操作，有可能涉及数据结构的复制。

我们当时考虑了两种可能方式：

- 1) 对使用左值和使用右值分别定义函数。
- 2) 要求程序员使用一个辅助性的数据结构。

最后还是选择了后一种方式，因为它能避免语言扩充，也因为我们认为在更普遍的情况下，所需要的技术就是返回一个对象，指明某个容器(例如String)里的一个位置。基本想法是构造一个辅助类，使它能像引用似地标识出容器里的一个位置，但却能对读和写分别提供语义。例如：

```

class char_ref { // identify a character in a String
friend class String;
    int i;
    String* s;
    char_ref(String* ss, int ii) { s=ss; i=ii; }
public:
    void operator=(char c);
    operator char();
};

```

对char_ref的赋值被实现为对被引用字符的直接赋值，而从char_ref的读操作则被实现为一个到char的转换，返回被指定的字符的值：

```

void char_ref::operator=(char c) { s->r[i]=c; }
char_ref::operator char() { return s->r[i]; }

```

注意，只有String可以创建char_ref。实际的赋值由String实现：

```

class String {
    friend class char_ref;
    char* r;
public:
    char_ref operator[](int i)
    { return char_ref(this, i); }
    // ...
};

```

有了这些定义，

```
s1[i] = s2[j];
```

的意思就是

```
s1.operator[](i) = s2.operator[](j)
```

这里s1.operator[](i)和s2.operator[](j)都返回类char_ref的一个临时对象，这转而又意味着

```
s1.operator[](i).operator=(s2.operator[](j).operator char())
```

通过在线化，就能使这种技术在大多数情况下都可以被接受，而用友关系限制char_ref的创建，能保证我们不会遇到临时对象的生存时间问题（6.3.2节）。举例来说，这种技术被成功地用于实现了一些String类。当然，对那些简单访问个别字符之类的使用而言，这种技术看起来还是复杂了一点，也太沉重了。因此我也一直在考虑其他方法。特别是一直在寻找一种效率更高的方法，又不带有为特殊用途而增加的累赘。组合运算符（11.6.3节）是一种可能性。

3.8 常量

在操作系统里，常常能见到用两个二进制位直接或间接地对一块存储区进行访问控制，用一个位指明某个用户能否在这里写，另一个指明该用户能否从这里读。我觉得这种思想可以直接用到C++里，因此也考虑过允许把一个类型描述为readonly或者writeonly。有一个内部备忘录 [Stroustrup, 1981b] 描述了这种想法，时间是1981年1月：

“直到现在，在C语言里还不能规定一个数据元素是只读的，也就是说，它的值必须保持不变。也没有任何办法去限制函数对传给它的参数的使用方式。Dennis Ritchie指出，如果readonly是一个类型运算符，我们就能很容易地得到这些能力。例如：

```

readonly char table[1024]; /* the chars in "table"
                           cannot be updated */

int f(readonly int * p)
{
    /* "f" cannot update the data denoted by "p" */
    /* ... */
}

```

这个readonly运算符可用于防止对某些位置的更新。它说明，对于所有访问这个位置的合法手段而言，只有那些不改变存储在这里的值的手段才是真正合法的。”

这个备忘录进一步指出：

“这种`readonly`运算符也可以用到指针上。`*readonly`被解释为‘对指向的只读’，例如：

```
readonly int * p; /* pointer to read only int */
int * readonly pp; /* read only pointer to int */
readonly int * readonly ppp; /* read only pointer
                           to read only int */
```

在这里，给`p`赋一个新值是合法的，但却不能给`*p`赋新的值；给`*pp`赋值是合法的，但不能给`pp`赋值；给`ppp`和`*ppp`赋值都是非法的。”

最后，这个备忘录还引进了`writeonly`：

“另外还可以有类型运算符`writeonly`，它的使用与`readonly`一样，但它是防止读而不是写，例如：

```
struct device_registers {
    readonly int      input_reg, status_reg;
    writeonly int     output_reg, command_reg;
};

void f(readonly char * readonly from,
      writeonly char * readonly to)
/*
    "f" can obtain data through "from",
    deposit results through "to",
    but can change neither pointer
*/
{
    /* ... */
}

int * writeonly p;
```

在这里的`++p`是非法的，因为它涉及到读`p`原来的值；而`p=0`则是合法的。”

这个建议所关注的是界面描述，而不是为C语言提供符号常量。很清楚，一个`readonly`值就是一个符号常量，但这个建议的范围却大得多。开始时我只提出了到`readonly`的指针，而没有提出`readonly`指针。与Dennis Retchie的一个简短讨论把这种想法发展为`readonly`/`writeonly`机制。我实现了这种机制，并把它提交给贝尔实验室内部的一个由Larry Rosler领导的C标准小组。这是我第一次在标准化方面的经验。我在离开一次有关会议时得到的是同意（通过投票）把`readonly`引进C语言（而不是带类的C或者C++），但将它另外命名为`const`。不幸的是这个决议并没有执行，因此我们的C编译系统里什么也没改变。后来ANSI C委员会（X3J11）成立了，有关`const`的建议又出现在那里，最后变成了ANSI/ISO C的一部分。

在此期间，我在带类的C里取得了许多有关`const`的经验，发现用`const`来表示常数，可以成为宏的一种有用替代物，但要求全局`const`隐含地只在它所在的编译单元起作用。因为只有在这种情况下，编译程序才能很容易推导出这些东西的值确实没有改变。知道了这些之后，我们就能把简单的`const`用到常量表达式里，并可以避免为这些常量分配空间。C语言没有采纳这个规则。例如，在C++里我们可以写：

```

const int max = 14;

void f(int i)
{
    int a[max+1]; // const 'max' used in constant expression

    switch (i) {
    case max:    // const 'max' used in constant expression
        // ...
    }
}

```

而即使是今天，在C语言里我们还是必须写：

```

#define max 14
// ...

```

因为在C语言里const不能用于常量表达式。这就使const在C语言里远没有在C++里那样有用。这也使C语言还要依赖于预处理程序，而C++程序员则能使用具有很好的类型和作用域的const。

3.9 存储管理

在写出第一个带类的C程序之前很久我就已经知道，在有类的语言里对自由存储区（动态存储区）的使用比大部分C程序里更频繁得多。这就是把new和delete运算符引进带类的C里的原因。new运算符从自由存储区中分配存储，并调用一个建构函数以确保进行正确的初始化，这种想法也是从Simula借来的。delete函数是必要的补充，因为我不希望带类的C依赖于一个废料收集系统（2.13节，10.7节）。有关new运算符的论据可以这样总结，你是希望写：

```
X* p = new X(2);
```

还是写

```

struct X * p = (struct X *) malloc(sizeof(struct X));
if (p == 0) error("memory exhausted");
p->init(2);

```

按哪种写法你更可能弄出错误呢？请注意，在两种情况下都要检查存储区耗尽的情况。使用new运算符时有一个隐含的检查，并可能调用用户提供的一个new_handler函数，见[2nd, 9.4.3节]。反对的论点——在那个时候这种声音是很多的——包括：“但是我们并不真正需要它”，“但是将来总会有人将new用作标识符”。当然这些意见也都对。

引进new运算符能使自由空间的使用更方便，更不容易出错。这样就能进一步扩大它的使用，这也使在许多实际系统里，用于实现new的C语言自由空间分配函数malloc()变成了执行的瓶颈。这不会令人感到吃惊，剩下的问题是如何对付这个问题。让实际系统把它们的50%或者更多的时间花在malloc()上也是无法让人接受的。

我发现采用按类的分配程序和释放程序效率很高。在这里的基本想法是，主导自由空间存储区使用的通常是很几个类的大量的小对象分配和释放操作。把这些对象的分配问题抽取出来，放到另一个独立的分配程序里，你就可以在这些对象上节约时间和空间，同时也减

少了通用自由空间里出现碎片的情况。

我已经不记得关于如何为用户提供这种机制的最早讨论的情况，但确实记得向Brian Kernighan和Doug McIlroy提出了“给this赋值”的技术（下面描述），并总结说“它丑陋得像是犯罪，但是还能用。如果你能想出一个能用的更好办法，我就会按你的方法做”，或者类似的话。他们没有做到，因此我们只好等到Release 2.0。现在的C++语言里已经有了另一种更清晰的解决办法（见10.2节）。

这里的想法是，一个对象的存储按默认方式是“由系统”分配的，不要求用户做任何特定操作。为了覆盖这种默认方式，程序员可以简单地通过给this指针赋值。按照定义，this指针正指向作为成员函数调用出发点的对象本身。例如：

```
class X {
    // ...
public:
    X(int i);
    // ...
};

X::X(int i)
{
    this = my_alloc(sizeof(X));
    // initialize
}
```

无论何时建构函数X::X(int)被调用，存储分配工作就将通过my_alloc()完成。这种机制对这里要做的事情以及一些其他情况都是足够强有力的，但就是太低级了，很难处理它与堆栈分配和继承的相互关系。这种机制也很容易包藏错误。如果一个重要的类里有很多建构函数（这种情况很典型），那么就需要重复地写上面这样的东西。

请注意，静态的或者自动的（在堆栈上分配的）对象总是可能出现的，最有效的存储管理技术密切依赖于这样的对象。字符串类是个典型的例子。String对象常常被分配在堆栈上，这时就再不需要明确关注它们的存储管理问题，而它们所依赖的自由空间则由String的成员函数管理，与用户无关，也是用户不可见的。

这里所用的有关建构函数的描述方法在3.11.2节和3.11.3节中讨论。

3.10 类型检查

C++的类型检查规则也是由带类的C中取得的那些经验的结晶。所有函数调用都在编译时进行检查，可以通过函数声明中的特殊描述形式抑制对于最后的一些参数的检查。对于C语言的printf()而言这也是必需的：

```
int printf(const char* ...); // accept any argument after
                             // the initial character string

// ...

printf("date: %s %d %d\n", month, day, year); // maybe right
```

这里还提供了一些机制，以便缓和后退的征兆：许多C程序员在第一次遭遇严格类型检查时都

出现了这种想法。用省略号制止类型检查是其中最激烈的也是最不被推荐的东西。函数名字重载（3.6.1节）和默认参数 [Stroustrup, 1986]（2.12.2节）使人能给出这样的表象，好像一个函数可以有各种各样的参数表，但同时又不会损害类型安全性。

除此之外我还设计了流I/O系统，以说明即使是对I/O系统也没有必要去减弱类型检查（见8.3.1节）：

```
cout<<"date: "<<month<< ' '<<day<< " 19"<<year<< '\n' ;
```

是前面例子的一个类型安全的改版。

我当时把类型检查看作是一种具有实际意义的工具，而不是以它自身作为目标，现在也依然这样看。从根本上说，应该认识到，清除掉一个程序里的所有违反类型规则的东西，并不意味着结果程序就能是正确的，甚至不意味着这个程序不会因为某个对象的使用方式与其定义方式不符而瘫痪。例如，一个天电脉冲可能导致某个关键性的存储器二进制位的值改变了，这种方式不可能与程序语言的定义相符。把类型的不安全性与程序瘫痪等同起来，把程序瘫痪与灾难性的故障，例如飞机失事、电话系统崩溃或者原子能反应堆熔化等同起来，这是不负责任的和误导的。

认为会产生这种效果的人，实际上是错误地认为系统的可靠性依赖于它的所有部件。把一个错误归咎到整个系统的某个特定部件只是简单地确定错误。我们希望能够设计出这样的生命攸关的系统，使其中一个甚至多个错误也不会导致系统“瘫痪”。系统完整性的责任还是在构造这个系统的人，而不是系统的任何部分。特别地，类型安全性并不想代替测试，虽然它非常有助于使系统能为测试做好准备。为了某个特定的系统错误（即使是纯粹的软件错误）去抱怨程序设计语言，那完全是把问题搞混了，参见16.2节。

3.11 次要特征

在从带类的C到C++的转变过程中也加进了一些次要的特征。

3.11.1 注释

最容易看到的次要变化是引进了具有BCPL风格的注释：

```
int a; /* C-style explicitly terminated comment */
int b; // BCPL-style comment terminated by end-of-line
```

因为同时允许这两种注释风格，人们可以选用他们所喜欢的形式。从个人角度说，我喜欢用BCPL风格写一行的注释。由引进//形式直接带来的一个情况是我有时会出现愚蠢的错误，忘记写C注释的结束，或者发现我原来用于结束/*注释的三个多余字符有时使一行返到了屏幕的前面。我还注意到，对注释掉一小段代码来说用//比用/*更方便。

不久人们就发现，增加//也不是与C语言100%兼容的。比如存在下面的例子：

```
x = a//* divide */b
```

这在C++里的意思是x=a，而在C里是x=a/b。那时大部分C++程序员都认为这种例子没什么实际价值。现在人们也这样看。

3.11.2 建构函数的记法

把建构函数称作“new-函数”常常造成混乱，因此就引进了命名的建构函数。与此同时，

这个概念又得到了进一步扩充，允许将建构函数显式地用在表达式里。例如，

```
complex i = complex(0,1);

complex operator+(complex a, complex b)
{
    return complex(a.re+b.re,a.im+b.im);
}
```

形如complex(x,y)的表达式是显式地调用类complex的建构函数。

为了尽量减少新的关键词，我没有使用下面这样更明确的语法：

```
class X {
    constructor();
    destructor();
    // ...
};
```

而是选择了能反应建构函数使用形式的声明方式：

```
class X {
    X(); // constructor
    ~X(); // destructor (~ is the C complement operator)
    // ...
};
```

这也可能是过于轻巧了。

允许在表达式里显式调用建构函数被证明是一种很有用的东西，但它也是C++语法分析问题的一个主要根源。在带类的C里，new()和delete()函数默认的就是public。在C++里去掉了这种不规则情况，建构函数和析构函数都遵循与其他函数一样的访问控制规则。例如：

```
class Y {
    Y(); // private constructor
    // ...
};

Y a; // error: cannot access Y::Y(): private member
```

这带来了一些有用的技术，它们的基本思想就是通过把执行操作的函数隐蔽起来，达到控制操作的目的。参见11.4节。

3.11.3 ■化

在带类的C里，圆点除了用于描述从某个对象选择成员外，也用于描述类的成员。这带来了一些不太重要的混乱，也可以用它构造出带有歧义性的例子。考虑：

```
class X {
    int a;
public:
    void set(X);
};

void X.set(X arg) { a = arg.a; } // so far so good

class X X; // common C practice:
```

```

    // class and object with the same name

void f()
{
    // ...
    X.a; // now, which X do I mean?
        // the class or the object?
    // ...
}

```

为化解这种问题，引进了用`::`表示类的成员关系，而将`.`保留专门用于对象的成员关系。这样上面的例子就变成：

```

void X::set(X arg) { a = arg.a; }

class X x;

void g()
{
    // ...
    x.a; // object.member
    X::a; // class::member
    // ...
}

```

3.11.4 全局变量的初始化

我的一个目标是让用户定义类型具有与内部类型同样的可用性。过去在Simula里，我也经历过由于不能有类类型的全局变量而导致的性能问题。所以在C++里允许类类型的全局变量。这个决定带来一些重要的但多少有点意外的后果。考虑：

```

class Double {
    // ...
    Double(double);
};

Double s1 = 2;           // construct s1 from 2
Double s2 = sqrt(2);    // construct s1 from sqrt(2)

```

一般来说，这种初始化无法在编译时或者连接时做完，需要做动态的初始化。这种动态初始化在编译单元内部按照声明的顺序进行。对于出现在不同编译单元里的对象，没有规定初始化的顺序，只要求所有静态初始化都必须在开始动态初始化之前完成。

1. 动态初始化的问题

我曾经假设全局的对象都是很简单的，因而只需要做一些相对简单的初始化。特别是我曾期望一个全局变量的初始化依赖于另外的编译单元里的其他全局变量的情况是极少出现的。我当时简单地认为这种依赖性就是设计拙劣，因此就不打算为解决这种问题提供特殊语言机制，不想承担任何责任。对于很简单的例子，比如上面的那些例子，我的看法是对的。这种例子很有用处，也不会造成什么问题。不幸的是，我后来发现了另一些更有意思的动态初始化全局对象的应用。

一个库在其各部分能够被使用之前，常常需要执行一些操作。换个说法，一个库可能需

要提供一些对象，它们被假定是预先初始化好的，用户可以直接使用它们而不必首先去做初始化工作。例如，你不必去初始化C语言的stdin和stdout，C语言的启动例行程序已经为你做好了这些事情。与此类似，C语言的exit()能为你关闭stdin和stdout。这是一类很特殊的处理，其他库中没有提供与此等价的能力。在我设计I/O流库时，也希望能够与C语言的I/O机制相媲美，并不希望把某些专用的赘瘤引进C++中。这样，我就简单地让cout和cin依赖于动态初始化机制。

这种方式工作得很好，但是我要依赖一些实现细节去保证cout和cin能在用户代码开始运行之前构造好，并在用户最后的代码完成之后被撤销。其他实现者可能没有想这么多，或者是不够小心。结果导致人们发现自己的程序因为在cout建构之前被使用而导致内存卸载(dump core)，或者因为cout被过早撤销(或者刷新)而导致一些输出丢失了。换句话说，我们被顺序依赖性咬伤了，而我以前认为这是“不大可能出现的拙劣设计”。

2. 绕过顺序依赖性

幸好这个问题并不是不可逾越的。实际上存在着两种解决办法：一种最明显的方法是给每个成员函数加上一个第一次调用开关。这种方法依赖于一个被默认地初始化为0的全局数据。例如：

```
class Z {
    static int first_time;
    void init();
    // ...
public:
    void f1();
    // ...
    void fn();
};
```

把各个成员函数都写成下面的样子：

```
void Z::f1()
{
    if (first_time == 0) {
        init();
        first_time = 1;
    }
    // ...
}
```

这是非常令人讨厌的。如果函数很简单，例如就是输出单个字符的函数，这样做的开销也可能很可观。

在重做I/O流的设计时(8.3.1节)，Jerry Schwarz采用了这种方法的一个聪明的变形[Schwarz, 1989]，在一个<iostream.h>头文件里包含下面这样的东西：

```
class io_counter {
    static int count;
public:
    io_counter()
    {
        if (count++ == 0) /* initialize cin, cout, etc. */
    }
}
```

```

~io_counter()
{
    if (--count == 0) { /* clean up cin, cout, etc. */ }
}
};

static io_counter io-init;

```

现在，在每个包含了*iostream*头文件的文件里都创建了一个*io_counter*，对它初始化的作用就是增加了*io_counter*的值。当这件事第一次发生时，有关的库对象被初始化。由于库的头文件出现在库函数的任何使用之前，这就保证了正确的初始化。由于析构是按照与建构的相反顺序做的，这项技术也能保证在库的最后使用之后的清理工作。

这项技术以一种具有普遍意义的方式解决了顺序依赖的问题，代价是微乎其微的，只要求库的提供者多写几行高度程式化的代码。不幸的是，这样做带来的性能问题也可能很严重。在采用这种诡计的地方，大部分C++对象文件都包含着动态初始化代码（假定用的是常规连接程序），这也就意味着动态初始化的例行程序将散布在进程的整个地址空间里。在采用虚拟存储器的系统里，这又意味着在程序的初启阶段和最后的清理阶段，它的大部分页面都需要装进基本存储器。这可不是使用虚拟存储器的良好行为方式，可能使某些重要应用的启动拖延许多秒的时间。

某个实现者倡议采用的另一种简单解决办法是修改连接程序，将动态初始化的代码集合到一个地方。另外，只要一个系统不支持某种将程序动态装入基本存储器的操作，这种问题也就不会出现。但是，对于受到这种问题困扰的C++用户而言 [Reiser, 1992]，上面这些不过是敷衍人的安慰罢了。这种做法从根本上违背了C++的格言：一个特征不仅应该有用，而且应该能负担得起（4.3节）。这个问题能够通过增加一个特征而得到解决吗？从表面上看似乎不可能，因为无论语言设计者或者官方的标准化委员会都不会为效率而立法。我看到过的建议都是针对顺序问题提出的——该问题早已被Jerry的初始化诡计解决了——而不是针对它们所隐含的效率问题。据我猜测，真正的解决办法是要找到某种方式，去鼓励实现者避免动态初始化例行程序“蹂躏虚拟存储器”。已经知道一些能达到这个目的的技术，但是在标准里明显地写出一些东西，对于鼓励这些正确做法是很有必要的。

3. 内部类型的动态初始化

在C语言里，要初始化一个静态对象，只能采用稍微扩充了一点的常量表达式。例如：

```

double PI = 22/7;      /* ok */
double sqrt2 = sqrt(2); /* error in C */

```

但是C++允许用完全一般的表达式做类对象的初始化。例如：

```
Double s2 = sqrt(2); // ok
```

这样就使内部类型反而变成了“二等公民”，因为为类提供的支持已经发展到超过了为内部类型提供的支持。这种反常现象是很容易解决的，但有关能力直到Release 2.0才成为普遍能够使用的东西：

```
double sqrt2 = sqrt(2); // ok in C++ (2.0 and higher)
```

3.11.5 声明语句

我从Algol 68借来的一个概念是可以将声明写在需要它的任何地方（而不是必须在某些

块的头部)。这样我们就允许了一种“只做初始化的”或说是“单赋值的”程序设计风格，这样做比传统风格更不容易出错。这种风格对于引用和常量而言更是根本性的，因为这些东西不能赋值。对那些采用默认初始化方式代价特别高的类型，这种机制从本质上说效率更高。例如：

```
void f(int i, const char* p)
{
    if (i<=0) error("negative index");
    const int len = strlen(p);
    String s(p);
    // ...
}
```

为尽量减少由未经初始化的变量带来的问题，我们另一方面的努力就是利用建构函数来保证初始化(2.11节)。

1. for语句里的声明

需要在块的中间引进新变量，最常见一个原因就是为循环提供一个变量。例如：

```
int i;
for (i=0; i<MAX; i++) // ...
```

为避免变量声明与它的初始化分离，我允许把声明移到for之后：

```
for (int i=0; i<MAX; i++) // ...
```

不幸的是，我没有抓住机会改变语义，将以这种方式引进的变量的定义域限制到for语句的范围内。忽视了这个问题的原因，从根本上说是为了避免给一般规则增加一种特殊情况，一般规则是“一个变量的作用域从它声明的点一直延伸到它所在块的结束位置”。

这个规则后来成为许多讨论的题目，最后也做了修改，以便与条件语句中声明的规则(3.11.5节)互相一致。也就是说，在一个for语句的初始化部分引进的名字的作用域在这个for语句的最后结束。

2. 条件语句里的声明

人们在谨慎地努力避免未初始化的变量，剩下的还有：

1) 用于输入的变量：

```
int i;
cin>>i;
```

2) 在条件语句里使用的变量：

```
Tok* ct;
if (ct = gettok()) { /* ... */ }
```

在1991年设计运行时的类型识别机制时(14.2.2节)，我认识到，如果允许将声明作为条件，就可以消除掉产生未初始化变量的后一种情况。例如：

```
if (Tok* ct = gettok()) {
    // ct is in scope here
}

// ct is not in scope here
```

这个特征并不只是一种减少打字输入的小窍门，它也是局部化思想的一个直接推论。通过把一个变量的声明、它的初始化以及对初始化结果的测试组合到一起，我们就得到了一种紧凑的表达方式，它能帮我们清除由于变量没有初始化就被使用而产生的错误。通过把这种变量的作用域限制到有关条件所控制的语句里，我们也解决了重新赋予这些变量其他用途的问题，以及在已经认为它们不再存在之后又不经意地使用的问题。这也就清除了另一个较小的错误根源。

允许在表达式里写声明的灵感来自表达式语言——特别是Algol 68。我“记得”Algol 68的声明产生值，所以就基于这点做了我的设计。后来我才发现自己的记忆是错误的，声明实际上是在Algol 68里仅有的几种不产生值的结构之一。我向Charles Lindsey提出这个问题，得到的回答是“甚至Algol 68也有些瑕疵，在某些地方它并不是完全正交的”。我想这不过是证明了一个语言并不必无愧于它自己的理想，也照样可以为人提供灵感。

如果我是从一张白纸出发去设计语言，我将遵循Algol 68的方式，让每个语句或者声明都是表达式，都产生一个值。我将禁止未初始化的变量，并抛弃在一个声明里说明多个变量的思想。当然事情很清楚，这些想法已经远远超出了能够被C++接受的范围。

3.12 与经典C的关系

由于引进了C++这个名字，写出了C++的参考手册 [Stroustrup, 1984]，与C语言的兼容性问题就变成了最重要的问题，也成为了争论的焦点。

还有，到了1983年后期，贝尔实验室里一个负责开发和支持UNIX、生产AT&T的3B系列计算机的分支机构开始对C++感兴趣，它已经希望为C++工具的开发投入一些资源。对于使C++的发展由一个人的独舞转变为一个公司支持的关键性项目所用的语言，这种发展确实是非常必要的。不幸的是，这同时意味着在开发管理层也要考虑C++了。

从开发管理层发出的第一个命令就是要求与C的100%兼容性。与C语言兼容的想法是非常明显的，也是很合理的。但程序设计的现实则不那么简单。作为开始，C++到底应该与哪个C兼容？到处都是C语言的方言，ANSI C虽然已经开始出现，但是要得到它的一个稳定版本还需要时日。ANSI C的定义也同样允许方言的存在。我记得那时计算过——不过是作为玩笑——存在大约3⁴²个严格符合ANSI C标准的方言。为得到这个数字，基本方法就是拿出所有未定义的或要求实现去定义的方面，用它作为指数，算式的底则采用不同可能性的平均数目。

很自然，普通用户所希望的与C兼容指的是C++与他的局部C方言兼容。这是一个很重要的实际问题，也是我和我的朋友特别关注的。业界的经理或者销售商对于这方面的关心就差多了，他们或者是对技术细节不甚了了，或者不过是想用C++把用户绑到自己的软件和/或硬件上。而贝尔实验室的C++开发者们则不同，他们独立于自己为之工作的机构，“把从感情上承担起兼容性的义务作为一个概念 [Johnson, 1992]”，并抵抗着管理层的压力，设法把一种C的方言隐藏进C++的定义中。

兼容性问题的另一个方面是更要紧的：“C++应该以什么方式与C不同，以便能达到它自己的目标？”还有“C++应该以什么方式与C兼容，才能达到它的目标？”问题的这两个方面同样重要，从带类的C转变到C++ Release 1.0的过程中，在这两个方向上都做了一些修正。很缓慢地，也是充满痛苦地，一个共同的意见逐渐浮现出来了：在C++和ANSI C（当它成为标准后）之间不应该存在无故的不兼容性 [Stroustrup, 1986]，而确实应该有一些不兼容性，只

要它不是无故的。很自然，“无故的不兼容性”这个概念成为许多争论的话题，它耗费了我太多太多的时间和精力。这个原则后来被广泛理解为“C++：尽可能地与C靠近，但又不过分地近”，这是到了Andrew Koenig和我一篇以此为名的文章[Koenig, 1989]之后。这种策略是成功的，一个标志就是K&R2 [Kernighan, 1988] 里的所有例子都是用C++的C子集写出来的。Cfront就是用来做K&R2里例子代码的基本测试所使用的编译程序。

关于模块化和如何通过组合一些分别编译部分的方式做出程序，在最初的C++参考手册[Stroustrup, 1984]里已经有明确的反映：

- (1) 名字是私用的，除非显式地将它们声明为公用的。
- (2) 名字局部于它们所在的文件，除非显式地从文件里引出。
- (3) 总是进行静态的类型检查，除非显式地抑制这种检查。
- (4) 一个类是一个作用域（这意味着类可以正确地嵌套）。

观点(1)不会影响与C的兼容性，但是(2)、(3)和(4)却隐含着不兼容性：

- 1) 按照默认方式，C的非局部函数和变量在其他编译单位里是可以访问的。
- 2) 在使用之前不必有C函数的声明，按照默认方式，C函数的调用不检查类型。
- 3) 在C语言里结构的名字不能嵌套（即使它们在词法上嵌套）。

此外，

- 4) C++只有一个名字空间，而C语言对“结构标志”有单独的名字空间（2.8.2节）。

这种“有关兼容性的战争”现在看起来是琐碎而无趣的，但还是留下一些基础性问题，仍然没有解决，我们还在ANSI/ISO标准化委员会里为它们而斗争。我非常强烈地认为，使兼容性战争发生并使它出奇地范围广泛的原因，在于我们从来没有直面关于C和C++语言不同目标的深刻问题，而一直把兼容性看作一些单独的需要个别解决的问题。

典型地，最不具根本性的问题4——“名字空间”花了最多的时间，最后是通过[ARM]的一种妥协解决了。

在把类作为作用域的概念（3）上我也不得不做些折衷，在发布Release 1.0时接受了C的“解决办法”。我原来一直没认识到的一个实际问题是，在C语言里一个struct并不构成一个作用域，因此像下面这样的例子：

```
struct outer {
    struct inner {
        int i;
    };
    int j;
};

struct inner a = { 1 };
```

在C语言里完全是合法的。不仅如此，这样的代码甚至可以在标准的UNIX头文件里找到。当时，这个问题在有关兼容性的斗争接近结束时被提了出来，我已经没有时间再去领会有关的C语言“解决方案”到底会带来什么，表示同意比与之斗争要容易得多。后来，在遇到了许多技术问题，并且接到来自用户的许多不满之后，嵌套的类作用域在1989年才被重新引进C++ [ARM]（13.5节）。

在经过了许多激烈辩论之后，C++对于函数调用的强类型检查才被接受了（没有修改）。对静态类型系统的隐含破坏是C/C++不兼容性的一种根本性例子，但是这绝不是无故的。

ANSI C委员会在这个问题上采纳了一种比C++规则和概念稍微弱一点的方式，并声明说：那些不符合C++的用法是过时的用法。

我不得不接受C的规则，在默认情况下，全局名字在其他编译单位里也是可以访问的。因为没有人支持更严格的C++规则。这也就意味着在C++里（与C类似），在类和文件的层次之上再也没有有效的模块化描述机制了。这导致了一系列的指责，直到ANSI/ISO委员会接受了名字空间（第17章）作为避免名字污染的机制。Doug McIlroy和其他人争辩说，不管怎样，如果在一个语言里，要想让一个对象或者函数可以从其他编译单元里访问就必须显式地声明，C程序员是不可能接受这个语言的。他们在当时可能是对的，也使我避免了一个大错误。我现在也认识到，原来的C++解决方案也不是足够优美的。

与兼容性有关的另一个问题是在这里总能看到两条战线，人们各自都对自己的观点坚信不移，认为必须为自己提出的情况辩护。一条战线要求100%的兼容性——常常并没有理解这样做实际上意味着什么。例如，许多要求100%兼容性的人在了解到这实际上意味着与现存C++的不兼容性，并将导致千百万行C++代码无法编译时，他们会感到大吃一惊。在许多情况下，强调100%兼容性的基本假设是C++只有不多的用户。另一种情况也常常见到，强调100%兼容性的后面隐藏着的是一些人对C++的忽视和对新特征的反感。

另一条战线也可能同样使人烦恼，他们声明与C的兼容性根本不是应该考虑的问题。他们为一些新特征辩护，而这些特征对于那些希望能混合使用C和C++代码的人而言将带来严重的不便。很自然，每条战线提出的更极端的论点都使另一战线进一步绷紧了神经，更加害怕损失掉自己最关心的语言部分。在这里——几乎总是——那些冷静的头脑占了上风，在所涉及的实际需要和使用C/C++的实际情况被认真地考虑过之后，争论通常收敛到对折衷细节的更具建设性的考察上。在X3J16 ANSI委员会的组织会议上，Larry Rosler，原来ANSI C委员会的编辑，对抱怀疑态度的Tom Plum解释说，“C++就是我们想做但却无法做成的那个C语言”。这可能是有点夸大其词，但是对于C和C++的共同子集而言，这个说法与真理相距并不远。

3.13 语言设计的工具

比黑板更高级的理论或工具对于C++的设计和进化并没有起过多少作用。我曾试图用YACC（一种语法分析程序的生成器，[Aho, 1986]）做语法方面的工作，但却被C语言的语法打败了（2.8.1节）。我考虑了指称语义学，但又被C的诡计所击退。Ravi Sethi曾经考查过这个问题，他发现无法用这种方式表述C语言的语义 [Sethi, 1980]。

最主要的问题是C的不规范性，以及有关C实现的一些依赖于具体实现的东西和未加规定的方面等。在很久以后，ANSI/ISO C++委员会曾请一些形式化定义的专家来解释他们的技术和工具，并说明他们的观点：在定义C++的标准化问题上，真正的形式化途径究竟有可能在多大程度上帮助我们。我也考查了ML和Module-2的形式化规范，想看看形式化途径是否可能给出一个比传统英文文字更短和更优美的描述。我不认为一个这样的C++描述被实现者或者专家用户做出错误解释的可能性会更少。我的结论是，一个语言的形式化定义如果不是和某种形式化定义技术一起设计的，那就将超出了所有人的能力范围，除了很少的几个形式定义的专家之外。我在那时的结论得到了证实。

但是，放弃对形式化规范的期望又使我们陷身于不准确和不充分的术语的包围之中。在

这种状况下，我能用什么作为补偿呢？我试着去对新特征做推理，自己做，也请别人检查我的逻辑。但是我不久发展出一种对于争论的健康的不敬观点（当然也包括对我自己），因为我发现对每个特征都可能构造出一个似乎有道理的争论。从另一方面看，如果你接受了能使某些人活得更方便的所有特征，结果得到的绝不会是一个好语言。现存的合理特征太多了，任何语言都不可能即提供所有这些东西，而又能具有内在的一致性。因此，只要可能我就设法去做试验。

不幸的是你也常常无法推导出正确的试验。要想提供一个完全规模的系统，带有实现、工具、教育，并且有些人用这个，另一些人用的是其他东西，使人能够衡量它们的差异，这显然是完全不可能的。人的差别太大，项目的差异太多，而所建议的特征在定义、实现和解释它们的努力中都会发生变化。所以我只好用定义、实现和解释特征的努力作为一种设计辅助。一旦实现了某个特征，我和另外几个人就会去使用它。我尽可能地去试，对做出任何正面的论断保持高度的疑心。只要可能，我就依靠那些正在考虑实际应用的老练程序员的观点。这样我就设法补偿了自己“试验”中最根本的局限性：这些试验通常只是比较各种实现方式，对小的实例检查源代码的质量，以及对这些实例运行时间和空间进行度量，等等。在这种设计过程中我至少能有一些反馈，使得我能够依靠试验，而不是仅靠纯粹的思维。我的牢固树立是，语言设计并不是纯粹的思维训练，而是一种在需要、想法、技术和约束条件之间取得平衡的非常实际的修炼。一个好语言不是设计出来的，而是成长起来的。这种修炼与工程、社会学和哲学的关系比与数学的关系更密切些。

回首往事，我希望当时能够知道一种方法，用它能形式化地描述类型转换和参数匹配。人们已经认识到，这类问题是很难做好的，也很难写出无歧义的文档。不幸的是，我怀疑现在也没有合理的通用方式，能以某种方便的方式处理C语言对内部类型和运算符的非常不规范的规则。

对语言设计者总存在着极大的诱惑，要他去提供某些特征或者服务，而在这些地方用户原来需要采用迂回的方式去解决问题。由于要求添加某些要求被拒绝而产生的尖叫声，远比对“又增加了另一个无用特征”的抱怨声响亮得多。对于标准化委员会，这也是一個很严重的问题（6.4节）。这种争论的最坏变形就是对正交性的崇拜。许多人认为，如果在语言中增加了一个特征可以使它更具备正交性，那么这就是接纳这个特征的充分论据。我赞成正交性在原则上是一种好东西，但也注意到它总要带来代价。在正交性的所有好的含义之外，在手册和指导材料中定义各种特征的组合通常也需要做许多额外工作。经常遇到的情况是，实现由正交性思想所规定的组合，实际上比人们想象的要困难许多。对于C++，我总在考虑正交性对那些不使用组合方式的人在运行时间和空间上的代价如何。如果这种代价无法在原则上弄成0，我就很不情愿接受这个特征——即使它是正交的。也就是说，正交性应该作为第二位的原则——放在对于有用性和效率的最基本考虑之后。

我过去和现在的印象都说明，许多程序设计语言和工具是在提供了解答之后再去寻找问题。而我则确定自己的工作决不能混同于这一类东西。这样，我关注着程序设计语言文献以及有关程序设计语言的争论，针对我的同事和我自己在实际应用中遇到的问题，寻找解决问题的想法。其他程序设计语言构筑起一座座思想和见解的大山，但它们都需要细心地挖掘，以避免特征泛滥和不一致性。C++的主要思想源泉是Simula、Algol 68，随后是Clu、Ada和ML。良好设计的关键是对问题的深入认识，而不是提供了多少最高级的特征。

3.14 《C++程序设计语言》(第1版)

在1984年秋天，我在工作上的邻居Al Aho建议我写一本有关C++的书，基于自己的文章、内部备忘录和C++用户参考手册，在结构上可以参照Brian Kernighan和Dennis Ritchie的《C程序设计语言》[Kernighan, 1978]。完成这本书花了9个月的时间。我在1985年8月中写完了这本书，第一个拷贝在10月中出来了。感谢美国出版界的好管闲事，这本书具有1986年的版权。

在书的前言里列出了到那时为止为C++做出了最大贡献的人：Tom Cargill, Jim Coplien, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Larry Rosler, Jerry Schwarz, 以及Jonathan Shopiro。我对于将一个人加入这个表的准则是：能够确定某个特定C++特征是由于这个人的提出而加进去的。

该书的开篇语是“C++是一种通用程序设计语言，其设计就是为了使认真的程序员能够觉得编程变得更愉快了。”这句话被审阅者删掉了两次，他们拒绝相信程序语言的设计除了对生产率、管理和软件工程的那些严肃的唧咕声之外还能够有什么。当然，

“C++原本的设计就是为了使作者和他的朋友们能够不必再用汇编语言、C或者各种各样新潮的高级语言做程序设计。它的主要目标是使程序员个人能够更容易和更愉快地写出好的程序来。”

实际情况就是这样，无论审阅的人愿不愿意相信。我把工作的注意力集中于人，个人（无论他是否在一个小组里），程序员。这种思考问题的方式随着时间的推移而逐渐加强，在第2版里更加突出[2nd]，在那里更深入地讨论了设计和软件开发问题。

对于不计其数的程序员而言，《C++程序设计语言》就是C++语言的定义和C++的导引。这本书的展示技术和结构（这是我带着感激之情从《C程序设计语言》那里借来的，可能并不总是很有技艺）已经成为数量惊人的文章和书籍的基础。写它的时候我带着强烈的决心，不准备去鼓吹任何特殊的程序设计技术。基于同样的想法，我也害怕由于忽视和家长作风的误导而给语言构筑进一些限制。我不希望这本书变成一篇有关自己个人爱好的宣言。

3.15 有关“什么是”的论文

在发布了Release 1.0并将书的影像拷贝送给印刷厂后，我终于有时间去考虑更大的问题，写写整体的设计论点了。正在这时，Karel Babcicky（Simula用户协会(ASU)的主席）从奥斯陆打来电话，邀请我于1986年在Stockholm的ASU会议上做一个关于C++的报告。我自然很想去，但又担心在Simula会议上展示C++很可能被看作自我吹嘘的一个庸俗范例，或是企图从Simula那里偷走用户。最后我说，“C++并不是Simula，为什么Simula用户希望听到它的情况。”Karel回答说，“啊，我们并不吊在语法形式上。”这就给了我一个机会，不仅写下C++是什么，还包括它被假定是怎样的以及在哪些地方它并没有符合这个理想。结果就是文章“什么是‘面向对象的程序设计？’”（*What is "Object-Oriented Programming"*）[Stroustrup, 1986b]。一个扩充版本发表在1987年6月在巴黎召开的第一次ECOOP会议上。

这篇文章的重要性在于它第一次揭示了作为C++支持目标的一组技术。所有以前的阐述为了避免不诚实或被看作是一种宣传，总是限制在描述已经实现并已经使用的特征上。这篇有关“什么是”的论文则定义了一组我认为支持数据抽象和面向对象程序设计的语言应该解决

的问题，并给出了所需要的语言特征的范例。

结果是重申了C++的“多范型”性质的重要性：

“面向对象的程序设计是利用继承机制的程序设计。数据抽象是使用用户定义类型的程序设计。除了少许例外，面向对象的程序设计将能够而且应该支持数据抽象。这些技术需要有效的正确支持。数据抽象从本质上需要在语言特征的形式上得到支持，而面向对象的程序设计则更进一步地需要得到程序设计环境的支持。为了达到通用性，支持数据抽象或者面向对象程序设计的语言又必须能有效地利用传统硬件。”

在这里也特别强调了静态类型检查的重要性。换句话说，C++在继承模型和类型检查方面遵循的是Simula而不是Smalltalk的路线：

“一个Simula类或者C++类为一集（其所有派生类的）对象描述了一个固定界面；而一个Smalltalk类则是为（其所有子类的）对象描述出一个初始的操作集合。换句话说，一个Smalltalk类是一个最小描述，用户可以自由地去试用没有在这里描述的操作；而一个C++类则是个准确的描述，用户得到一种保证，只有在类声明中描述的操作才能被编译程序所接受。”

这将对人们设计系统的方式、对语言到底需要哪些设施产生深刻的影响。动态类型语言如Smalltalk能够简化库的设计与实现，其方式是把类型检查推迟到运行之中。例如（这里用了C++的语法形式）：

```
void f() // dynamic checking only, not C++
{
    stack cs;
    cs.push(new Saab900);
    cs.pop()->takeoff(); // Oops! Run-time error:
                          // a car does not have a
                          // takeoff method.
}
```

这种延迟检查类型错误的方式在C++里被认为是不可接受的。在一个动态类型的语言里，必须有一种完成记法规则与标准库匹配的方式。带参数类型为此问题提供了一种C++的（未来的）解决办法：

```
void g()
{
    stack(plane*) cs;

    cs.push(new Saab37b); // ok a Saab37b is a plane
    cs.push(new Saab900); // error, type mismatch:
                          // car passed, plane* expected.

    cs.pop()->takeoff(); // no run-time check needed
    cs.pop()->takeoff(); // no run-time check needed
}
```

把这类问题的编译时检查看成最重要的事情，一个根本原因就是看到了C++常常被用在运行时并没有程序员待在旁边的程序方面。从根本上说，静态类型检查的概念被看成是为程序提供一种尽可能强的保证，而不仅仅是作为一种取得运行效率的手段。

部分地说，这是有关机器能保证那些东西的普遍概念的一种特殊情况，也来源于人和排错工作不应该做什么的普遍性规则。很自然，静态类型检查对于排错很有帮助。但是，把基础放在静态类型检查上的最根本原因，则是因为我那时就确信（现在依然），一个由通过了静态检查的部件组合而成的程序，与一个基于弱类型检查或动态类型检查的界面的程序比起来，更可能是忠实地表达了一种经过深思熟虑的设计。当然，也应该记住，并不是对每个界面都能彻底地做静态类型检查，静态类型检查也绝不意味着不会有错误。

这篇有关“什么是”的文章列出了C++存在的三方面缺陷：

- 1) Ada、Clu和ML都支持带参数的类型，而C++却不支持。在这里使用的语法是为了说明问题而简单设计的。在需要的地方，带参数的类可以利用宏来“冒充”。很清楚，带参数的类在C++里将是特别有用的。编译程序很容易处理它们，但是当前的C++程序设计环境还没有复杂到能够支持这种功能，而又不带来太大的开销和/或不方便性。与直接描述类型相比，在这里不应该有任何多余的运行开销。
- 2) “随着程序越来越大，特别是当程序库被广泛使用时，处理错误（用更普遍的说法：“异常情景”）的标准将变得日益重要起来。Ada、Algol 68和Clu都有各自的支持某种处置异常的标准方式。不幸的是，在C++里还没有这种东西。在需要时，异常可以用函数指针、“异常对象”、“错误状态”以及C标准库的signal和longjmp等机制来“冒充”。一般地说，这是不能令人满意的，应该提供一种处理错误的标准框架。”
- 3) “有了这种解释，事情变得很明显，让一个类B可以从两个基类A1和A2继承应该是很有用的，这称为多重继承。”

所有这三种机制都与提供更好的（也就是说更一般的和更灵活的）库有联系。所有这些现在在C++里都可以用了（模板，第15章；异常，第16章；多重继承，第12章）。请注意，增加多重继承和模板功能早就被考虑作为进一步发展的可能方向 [Stroustrup, 1982b]。在这篇文章也提出了异常作为另一个可能性，但我那时主要是担心，而不是正面地提出向这个方向发展的可能需求。

与往常一样，我又提出了运行时间和空间的效率要求，以及如果“不是完美的”就需要与其他语言在传统系统中共存的能力。对于一个声称自己为“通用”的语言，当然不应该违背这些东西。

第4章 C++语言设计规则

如果地图与地表不符，
要相信地表。

——瑞士军队格言

C++设计规则——整体设计目标——社会学规则——C++作为一种支持设计的语言——语言的技术性规则——C++作为一种支持低级程序设计的语言

4.1 规则和原理

要成为真正有用的东西，一个程序设计语言的设计就必须有一种全局观点，用它来指导其各个语言特征的设计。对于C++，这种全局观点由一组规则和约束构成。称它们为规则，是因为我认为原理这个词在一个真正科学原理非常贫乏的领域中显得过于自命不凡了，而程序设计语言设计就是这样的一个领域。此外，对许多人而言，术语原理也意味着一种不实际的推论，说任何例外都是不能接受的。我的有关C++设计的规则几乎可以保证都有例外情况。实际上，如果一条规则与某个实际试验发生冲突，这个规则就应该靠边站。这样说看起来似乎有些粗鲁，但是它不过是一般原则的一个变形：理论必须与试验数据相吻合，否则就应被更好的理论取代。

这些规则绝不能不假思索地使用；也不能用几条肤浅的口号取代。作为一个语言设计者，我把自己的工作看成是去决定需要对付的是哪些问题，决定C++的框架里能够对付的是哪些问题，并在实际语言特征设计的各种规则之间保持一种平衡。

这些规则指导着与语言特征有关的工作。当然，改进设计的大框架是由C++的基本设计目标提出来的（见下表）。

目 标
C++应该使认真的程序员能够觉得编程变得更愉快
C++是一种通用的程序设计语言，它应该
——是一种更好的C
——支持数据抽象
——支持面向对象的程序设计

我把有关规则组织在下面四个更具概括性的小节里。4.2节包含了有关整个语言的思想。这里的东西非常具有普遍性，个别的语言特征将无法直接放进这个画面里。4.3节的一组规则基本上是关于C++在支持设计方面所扮演的角色。4.4节的一组规则是关于语言形式的技术情况。而4.5节的一组规则集中关注C++作为低级系统程序设计语言所扮演的角色。

这些规则的形式主要得益于事后的思索，但有关的规则和所表述的观点在1985年C++的第一个发布之前就已经支配着我的思想了，而且——如前面章节里讲的——这些规则中的不少还是带类的C初始概念的一部分。

4.2 一般性规则

最一般和最重要的C++规则与语言的技术方面并没太大关系。这些规则几乎都是社会性的，其关注点是C++所服务的社会团体。C++语言的性质在很大程度上出于我的选择，我认为它们应该服务于当前这一代系统程序员，支持他们在当前的计算机系统上解决当前的问题。最重要的是，当前这个词的意义和性质总在随着时间而变化，C++必须能发展，以满足它的用户的需要；它的定义不应该是一成不变的东西。

一般性规则

- C++的发展必须由实际问题推动
- 不被牵涉到无益的对完美的追求之中
- C++必须现在就是有用的
- 每个特征必须存在一种合理的明显实现方式
- 总提供一条转变的通路
- C++是一种语言，而不是一个完整的系统
- 为每种应该支持的风格提供全面支持
- 不试图去强迫人做什么

C++的发展必须由实际问题推动：在计算机科学中，就像在许多其他领域中一样，我们总能看到许多人在努力为他们最喜爱的解决办法寻找问题。我不知道有任何简单明了的方法能够避免时尚扭曲我对什么最重要的认识，但是我也敏锐地意识到，提供给我的许多语言特征在C++的框架里根本就是不可行的，常常是与真实世界的程序员无关的。

改变C++的正确推动力是一些互相独立的程序员证明了这个语言对于表述他们的工作项目存在哪些不充分的地方。我偏爱来自非研究性项目的信息。无论何时，只要可能，在努力发现问题和寻找解决办法时我总设法与实际用户联系。我如饥似渴地阅读程序设计语言的文献，寻找对于这些问题的解答，也寻找各种可能有所帮助的技术。但是我也发现，文献在考虑什么是真正的问题方面是完全不可靠的，理论本身并不能为加入或者去掉一个特征提供充分的证据。

不被牵涉到无益的对完美的追求之中：任何程序设计语言都不是完美的，由于问题和系统都在持续的变化之中，将来也不会有完美的语言。用许多年的功夫去修饰一个语言，以图去接近某种完美的概念，这样做只能是使程序员无法从那些年的进步中获益。这也会使语言的设计者不能得到真实的反馈。没有适当的反馈，一个语言就会逐渐发展成与时代不相关的东西。在不同环境里的问题、计算机系统、以及——最重要的——人都有极大的差异，因此对某些小环境的“完美配合”几乎可以确定必然是过分特殊的，与大千世界的繁荣没有多大关系。在另一方面，程序员实际上花费了他们的大部分时间去修改老代码，或者与它们接口。为了完成实际工作他们需要某种稳定性。一旦某个语言投入了实际应用，对它的剧烈修改就不可行了，甚至想做点小修改而又不伤害到用户也是很困难的。因此，重要改进的需求必须依靠反馈信息，并伴以对兼容性、转变过程和教育的认真考虑。随着语言逐渐成熟，人必须更多地倾向于使用基于工具、技术和库的替代性方式，而不是去改变语言本身。

并不是每个问题都需要用C++解决，也不是说C++中的每个问题都足够重要，值得去做一种解决方案。例如，C++就没必要扩充去直接处理模式匹配或者定理证明；著名的C语言运算符优先级的毛病（2.6.2节）也最好让它待在那里，或者是通过警告信息去处置。

C++必须现在就是有用的：许多程序设计是很世俗的，在功能相对很差的计算机上做出来，运行在相对过时的操作系统和工具之上。大部分程序员没经过应有的形式化训练，几乎没有时间去更新他们的知识。为了能够为这些程序员服务，C++必须能适合具有平均水平的人，能用于平均水平的计算机。

虽然也多次想过尝试一下，但我从来没有真正的欲望去抛弃这些人以获得某种自由，去调整我的设计以满足最尖端的计算机和计算机科学的研究者们的口味。

这个规则的意义——与大部分其他规则一样——也将随着时间推移而改变，部分地也是C++成功的结果。最有威力的计算机今天已经可以用了，更多的程序员现在接受了C++所依赖的基本概念和技术。进一步说，随着人们抱负和期望的增长，程序员所面对的问题也改变了。这也意味着要求更多计算机资源，要求更成熟程序员的特征在今天已经可以并且应该考虑了。异常处理（第16章）和运行时的类型识别（14.2节）是这方面的例子。

每个特征必须有一种合理的明显实现方式：不应该有必须通过复杂算法才能正确有效实现的特征。最理想的是应该存在明显的分析和代码生成策略，而这些应该足够应付实际的使用。如果进一步思考能产生更好的结果，那当然是越多越好。大部分特征都通过实现、试验性的使用、检查修订，而后才被接受。在那些没按这种方式去做的地方，例如模板实例化机制（15.10节），后来就暴露出了问题。

当然，使用者总比写编译系统的人多得多，所以，如果出现在编译复杂性和使用复杂性之间的权衡问题，解决方案必定是偏向用户的。我在许多年做编译系统维护的生涯中，已经真正理解了这个观点。

总提供一条转变的通路：C++必须逐渐发展，以便能很好地服务于它的用户，并从反馈中获益。这隐含着必须特别关心保证老代码能继续使用的问题。当某种不兼容性已经无法避免时，就需要特别关注如何帮助用户更新他们的代码。类似地，必须提供一条路径，使人能够从容易出错的C那样的技术转到C++的更有效的使用方面来。

要清除一种不安全、容易出错、或者简单的就是有毛病的语言特征，最一般的策略就是首先提供一种更好的替代品，而后建议人们避免使用老的特征或者技术，而只有到数年之后——如果要做的话——再删除那个有问题的特征。这种策略可以有效地通过让编译产生警告信息的方式去支持。一般地说，直接删除一个特征或者更正一个错误是不可行的（典型原因是为了保持与C的兼容性）；替代的方法是提出警告（2.6.2节）。这样的C++实现就可能比仅根据语言定义看的情况更安全些。

C++是一种语言，而不是一个完整的系统：一个程序设计环境包含许多部分。一种方式是将多个部分组合为一个“集成化的”系统，另一种方式是维持系统中各个部分之间的经典划分，例如编译器、连接器、语言的运行支持库、I/O库、编辑器、文件系统、数据库，等等。C++遵循的是后一条路。通过库、调用的约定等，C++就能够适应于各种系统中指导语言和工具之间相互操作的有关系统规定。这对于移植和实现的简单性是非常关键的。更重要的是，这对于支持不同语言写出的代码之间相互操作也很关键。这种方式也能允许工具的共享，使作为个人的程序员能更容易地使用多种语言。

C++的设计是准备作为许多语言中的一个。C++支持工具的开发，但又不强求某种特定的形式，程序员仍然有选择的自由。关键的思想是，C++和与它关联的工具应该对给定的系统有一种正确的“感觉”，而不是给什么是一个系统或者一个环境强加上某种特殊的观点。

对于大型系统和带有某些特别约束的系统而言，这是非常重要的。这种系统常常不能得到很好的支持，因为“标准的”系统总是倾向于特别地去支持个人或者是很小的组，做相当“普通的”工作。

为每种应该支持的风格提供全面支持：*C++*必须发展，以满足严肃认真的开发者们的需要。简洁是最基本的，但也应该相对于那些将要使用*C++*的项目的复杂性来考虑这个问题。与保持语言定义比较短相比，应该认为，用*C++*写出的系统的可维护性和运行时的性能是更重要的问题。这实际上意味着一个相对比较大的语言。

这也意味着——正如许多经验所说明的——必须支持各种风格的程序设计。人们并不是只写那种符合狭义的抽象数据类型或面向对象风格的类，他们也写同时具有两方面特点的类，这样做通常都有很好的理由。他们还会写出这样的程序，其中的不同部分使用不同风格，以适应具体需要或者个人的口味。

因此，应该把语言特征设计成能够以组合方式使用，这就导致了*C++*设计中相当程度的正交性。支持“非正常”使用的可能性对于灵活性有很高的要求，这也已经一再导致*C++*被用到一些领域中，在那里一个更受局限的狭隘语言可能早就失败了。例如，*C++*中有关访问保护、名字检索、*virtual* /*nonvirtual*约束和类型的规则是相互独立的，这就打开了一种可能性，使人们可以同时使用依赖于信息隐蔽和派生类的各种技术。有些人愿意看到语言只支持很少的几种程序设计风格，从而认为这里的做法是黑客手段。在另一方面，正交性并不是第一原则，只有在它不与其他规则冲突，既能提供某些利益而又不使实现复杂化的时候，我们才采纳它。

提出一种相对较大的语言，这也意味着在管理复杂性方面的努力需要有些转移，从库和个别程序的理解方面转到学习语言和它的基本设计技术。对大部分人而言，这种重点转移、对新程序设计技术的采纳、对于“高级”技术的使用都只能逐步完成。很少人能够“一蹴而就地”完全掌握新技术，或者一下就能把所有的新招都用到自己的工作里（7.2节）。*C++*在设计中就考虑了如何使这种渐变成为可能的和自然的。这里的基本思想是：你不知道的东西不会伤害你。静态类型系统和编译的警告信息在这方面非常有用。

不试图去强迫人做什么：程序员都是很聪明的人，他们涉足于挑战性的工作，需要获得所有可能的帮助，不仅是其他的支撑工具和技术，也包括程序设计语言。试图给程序员强加严格的限制，使他们“只能做的正确事情”，从本质上说是搞错了方向，也一定会失败。程序员总能找到某种方法，绕过他们觉得无法接受的规则和限制。语言应该支持范围较广泛的设计和编程风格，而不应该企图去强迫程序员采纳某种惟一的写法。

这并不意味着所有的程序设计方式都同样好，或者说*C++*应该支持所有种类的程序设计风格。*C++*的设计是为了直接支持那些依靠广泛的静态类型检查、数据抽象和继承性的设计风格。当然，关于应该使用哪些特征的训教被维持到了最小的程度。语言机制尽可能保持了一种自由的政策，没有专门为排斥任何一种确定的程序设计风格而向*C++*里加进或者从中减去一种特征。

我也很清楚地意识到，并不是每个人都赞赏选择和变化。无论如何，那些偏爱带有较多限制环境的人可以在*C++*里始终如一地坚持使用某些规则，或者另去选用一种只给程序员提供很小的一组选择的语言。

许多程序员特别反感被告之某种东西可能是个错误，而实际上它并不是。所以，“可能的

“错误”在C++里并不是一个错误。例如，写一个能允许歧义使用的声明本身并不是错误，错的是那些存在歧义性的使用，而不只是这个错误的可能性。按照我的经验，多数“潜在错误”根本不会显现出来，因此拖延错误信息的方式就是不把它给出来。这种拖延将带来许多方便和灵活性。

4.3 设计支持规则

在下面列出的规则，所讨论的基本上是关于C++在支持基于数据抽象和面向对象的程序设计方面所扮演的角色。也就是说，它们更多关心的是语言在支持思考和表达高层次的想法方面所扮演的角色，而不是按照C或者Pascal的方式作为一种“高级汇编语言”所扮演的角色。

设计支持规则

- 支持一致的设计概念
- 为程序的组织提供各种机制
- 直接说出你的意思
- 所有特征都必须是能够负担的
- 允许一个有用的特征比防止各种错误使用更重要
- 支持从分别开发的部分出发进行软件的组合

支持一致的设计概念：任何个别的语言特征都必须符合一个整体模式，这个整体模式必须能帮助回答一个问题：什么能力是我们所需要的。语言本身不可能提供这种东西，这个指导模式必然来自另一个完全不同的概念层次。对C++而言，这个概念层次就是有关程序应该如何设计的基本思想。

我的目标是提升系统程序设计过程中的抽象层次，其方式类似于C语言在取代汇编语言作为系统工作的主流时的所做所为。有关新特征的想法都要放在统一的框架中考虑，看它们在将C++提升为一种表述设计的语言时能起什么作用。特别是对个别特征的考虑，要看它能否形成一种可以通过类进行有效表述的概念。这对于C++支持数据抽象和面向对象的程序设计是最关键的问题。

一个程序设计语言不是也不应该是一个完整的设计语言。因为设计语言应该是更丰富的，它不必像一个适合做系统程序设计的语言那样过多地关心细节。但是，程序设计语言也应该尽可能直接地支持某些设计概念，以便使设计师和程序员（这些人也常常是“戴着不同帽子的”同一批人）之间更容易沟通，并能简化工具的构造。

用设计的术语观察程序设计语言，就容易做到基于这种语言和它们所支持的设计风格之间的关系去考虑接受或者拒绝人们建议的语言特征。没有一种语言能支持所有的风格，而如果一个语言只支持某种定义狭隘的设计哲学，它也将因为缺乏适应性而失败。提升C++语言，以支持那些能够映射到“更好的”C / 数据抽象 / 面向对象的程序设计这样宽的谱中的各种设计技术，就能帮助我们避免企图把C++弄成对于所有人的惟一手段，也能对发展和进步提供始终如一的推动力。

为程序的组织提供各种机制：与C语言相比，C++能更好地帮助人们组织程序，使之更容易书写、阅读和维护。我把计算看成是一种已经由C语言解决了的问题。和其他几乎所有的人一样，我也有一些关于C的表达式和语句应该如何改进的想法。但是我决定把自己的努力集中到另外的方面。只要遇到关于某种新类型的表达式或者语句的建议，都需要仔细进行评价，

看它是能够影响程序的整体结构呢，还是仅仅使表达某种局部的计算更容易些。除了不多的几个例外，例如允许声明出现在真正需要变量的位置（3.11.5节）等，C语言的表达式和语句结构都保持不变。

直接说出你的意思：低级语言有一个最根本的问题，这就是在人们互相交流时能如何表述问题，和他们在使用程序设计语言时能如何表述问题之间存在一条鸿沟。程序的基本结构常常被淹没在二进制位、字节、指针和循环等的泥潭之中。

缩小这种语义鸿沟的最基本方法就是使语言更具有说明性。C++语言提供的每种机制都与使某种东西更具说明性有关，而后是为一致性检查、检测出愚蠢的错误、或者改进所生成的代码而开发出一些附加结构。

在无法使用说明性结构的地方，某种更明显的记法常常会有所帮助。分配/释放运算符（10.2节）和新的强制转换记法（14.3节）都是很好的例子。对于直接而明显地表达意图的想法，很早就有一种表述：“允许用语言本身表达所有重要的东西，而不是在注释里或者通过宏这类黑客手段。”这也就意味着，一般地说，这个语言必须具有比原来的通用语言更强的表达能力和灵活性，特别是它的类型系统。

所有特征都必须是能够负担的：仅仅给用户提供一种语言特征或者针对某个问题建议一种技术还是不够的。所提供的解决方案还必须是能负担得起的，否则这个建议简直就是一种侮辱，就像对于提问“什么是到孟菲斯的最好方式”，你回答说“去租一架专机”一样。对不是百万富翁的人，这绝不是一个有益的回答。

只有在无法找到其他方法能在付出明显更低的代价之下达到类似效果时，才应该把一个特征加进C++。我自己的经验是，如果程序员有在高效地或者优雅地做某种事情之间做选择的权利，大部分人将选择效率，除非存在其他更重要更明显的原因。例如，提供在线函数是为了无代价地跨过保护的边界，对宏的许多使用而言，这都是另一种具有更好行为的选择。这里的思想是显然的，一种功能应该同时是优雅的而又是高效的。在无法同时达到这些的地方，或者就不提供这种功能，或者是——如果要求很迫切——高效地提供它。

允许一个有用的特征比防止各种错误使用更重要：你可以在任何语言里写出很坏的程序。重要的问题是尽量减少偶然将某些特征用错的机会。我们花了许多精力，去保证在C++里各种构造的默认行为或者是有意义的，或者将导致编译错误。例如，按照默认方式所有函数的参数类型都做检查，即使是跨过了分别编译的边界；还有，按照默认方式，所有的类成员都是私用的。当然，一个系统程序设计语言不应该禁止程序员有意识地去打破系统的限制，所以设计的努力应该更多地放在提供机制，帮助人写出好的程序方面，而不是放在禁止不可避免的坏程序方面。在长期的过程中程序员必然会学习。这种观点也是C语言传统上“相信程序员”口号的一种变形。提供各种类型检查和访问控制规则，这就使类的提供者能够清楚地表述他对类的使用者期望什么东西，并提供保护以防偶然事故。这种规则并不是想提供一种保护去防止有意违反规则的情况（2.10节）。

支持从分别开发的部分出发进行软件的组合：复杂应用需要比简单程序更多的支持，大程序比小程序需要更多的支持，效率约束很强的程序比资源非常丰富的程序需要更多的支持。在第三个条件的约束之下，在C++语言的设计中花了很多精力去解决前两个问题。当实际应用变得更大更复杂时，这些应用必然是由一些人们能够把握的具有一定独立性的部分组合起来的。

任何能用于独立进行大系统的部件开发，而后又允许将它们不加修改地用到大系统里的东西都可以服务于这个目标。C++的许多发展都是由这个思想推动的。类本身从根本上说也就是这样的C++特征，抽象类（13.2.2节）能显式地支持界面与实现的分离。事实上，类可以用来表述一系列互相联系的策略 [Stroustrup, 1990b]。异常机制允许从一个库出发去处理错误（16.1节），模板使人能够基于类型进行组合（15.3节，15.6节，15.8节），名字空间解决名字污染的问题（17.2节），而运行时的类型识别处理这样一类问题：当一个对象被传递通过一个库时，其准确的类型有可能丢失，在这种情况下应该做什么？

程序员在开发大系统时需要得到更多的帮助，这还意味着不能由于过分依赖那些只对小程序有特效的优化技术而遭受效率损失。因此，对象的布局情况应该能在给定的编译单位内部孤立地确定，而虚函数调用也应该能编译成有效的代码，不依赖于跨越编译单位的优化。这确实做到了，甚至在高效意味着与C相比非常有效的意义下。在有关整个程序的信息能够使用时，再做进一步的优化也是可能的。例如，通过检查整个程序，一个对虚函数的调用——在不牵涉动态链的情况下——有时可以确定为一个实际函数调用。在这种情况下虚函数调用就可以用一个正常的函数调用取代，甚至用在线方式取代。能够做这种事的C++实现已经有了。当然这种优化对于生成高效代码并不是必需的，它们不过是在希望更高的运行效率，而不是在编译效率和动态链接新的派生类的情况下，可以获得的一些附加利益。当这种全局优化不能合理地完成时，虚函数调用还是能够通过优化去掉，只要这个虚函数是应用在已知类型的对象上，Cfront的Release 1.0就能做这件事。

对于大系统的支持经常是在“对于库的支持”的题目下讨论的（第8章）。

4.4 语言的技术性规则

下面的规则是针对这样的问题：事物如何在C++里表述。这里不讨论能够表述什么。

语言的技术性规则

- 不隐式地违反静态类型系统
- 为用户定义类型提供与内部类型同样好的支持
- 局部化是好事情
- 避免顺序依赖性
- 如果有疑问，就选择有关特征的那种最容易教给人的形式
- 语法是重要的（常以某些我们不希望的方式起作用）
- 应该消除使用预处理程序的必要性

不隐式地违反静态类型系统：每个对象在建立时就具有特定的类型，例如`double`，`char*`，`dial_buffer`等。如果以一种与对象的类型不一致的方式去使用它，那就是违背了类型系统。绝不允许这种情况发生的语言被称为是强类型的。如果一种语言能在编译的时候确认所有违反类型系统的情况，那么它就是强静态类型的。

C++从C语言继承了许多特征，例如联合、强制转换和数组，这就使它不可能在编译时检查出所有违反类型系统的情况。这也就是说，你需要显式地使用联合、强制转换、数组、显式的不加检查的函数参数、或者显式地使用不安全的C连接去违反这里的类型系统。所有这些不安全特征的使用都可以导致一个警告（在编译时）。更重要的是，现在C++已经拥有了一些语言特征，采用它们可以更方便而又同样有效，可以避免使用那些不安全的特征。这方面的

例子包括派生类（2.9节），标准数组模板（8.5节），类型安全的连接（11.3节），以及动态检查的强制转换（14.2节）。由于与C语言兼容性的需要以及常见的实践，维持着目前这种状态的路还很长也很困难，但大部分程序员已经采纳了那些更安全的方式。

只要有可能，总在编译时进行检查。只要有可能，那些在处理单独编译单位时只能提供信息而无法检查的东西都在连接时进行检查。最后，这里还提供了运行时的类型信息（14.2节）和异常机制（第16章），以帮助程序员处理编译和连接都无法捕捉到的错误条件。当然，在实践中还是编译时检查的代价更低，也更值得信赖。

为用户定义类型提供与内部类型同样好的支持：因为用户定义类型被当作C++程序的核心，它们当然需要从语言中得到尽可能多的支持。因此，例如“类对象只能在自由空间中分配”这样的限制就是无法接受的了。对于例如complex这样的算术类型，确实需要真正的局部变量，这也就导致了对面向值的类型（实在类型）的支持不但可以与内部类型媲美，甚至还超过了它们。

局部化是好事情：在写一段代码时，人们总希望它是自足的，除了在那些需要从其他地方得到服务的地方。人们也希望能使用一种服务又不带来过多的麻烦和干扰。而在另一方面，人们也需要为其他人提供函数、类等，而不必担心其实现细节与其他人的代码之间出现相互干扰的可能性。

C语言距离这些思想都非常遥远，连接程序可以看到所有全局函数和全局变量的名字，这些名字会与同样名字的其他使用互相冲突，除非把它们显式地声明为static。每个名字都可以当作函数名字使用而不必事先声明。作为早年把结构成员名也看成全局名字的一种遗风，在一个结构里面声明的结构也是全局的。此外，预处理程序的宏处理根本不考虑作用域问题，因此，只要改变了头文件或者编译程序的某些选项，程序正文中的任意一段字符都可能被改成另外的什么东西（18.1节）。如果你想去影响某些看起来是局部的代码，或者希望通过某些小的“局部”修改影响整个世界的其他部分，上面这些东西都是极其强有力的。公平地说，我认为这些东西对于理解和维护复杂的软件具有破坏性。因此我决心提供更好的隔离手段，以对抗从“其他地方”来的破坏，对于把什么东西从自己的代码中“引出”提供更好的控制。

对于代码局部化，使访问总通过定义良好的界面进行而言，类是第一位的最重要的机制。以后的嵌套类（3.12节，13.5节）和名字空间（第17章）又扩展了局部化范围的概念，明晰了访问的许可。在各种情况下，一个系统中的全局信息总量都显著地减少了。

访问控制使访问局部化，且没有因为完全的隔离而造成运行时间或存储空间的额外开销（2.10节）。抽象类使人可以以最小的代价得到最大程度的隔离（13.2节）。

在类和名字空间中人们可以将声明和实现分开，这也是非常重要的，因为这样能使人更容易看到一个类到底做了些什么，而不必不断地跳过描述有关工作如何完成的函数体。允许在类声明中写在线函数，这样，当上述分离不太合适时也可以达到另一种局部性。

最后，如果代码中重要的块都能放进一个屏幕，对于理解和操作也将大有裨益。C语言传统的紧凑性在这方面很起作用，C++允许在需要使用的地方引进新变量（3.11.5节），也是在这个方向上前进了一步。

避免顺序依赖性：对顺序的依赖性很容易使人感到困惑，在重新组织代码时也容易引进错误。人们都会注意到语句将按照定义的顺序执行，但是却往往忽视全局声明之间和类成员

声明之间的相互依赖性。重载规则（11.2节）和基类的使用规则（12.2节）都经过了特殊处理，使之能避免出现对顺序的依赖性。理想情况是，如果交换两个声明的顺序会导致另一种不同的意思，那么这就应该是一个错误。对于类成员的规则就是这样（6.3.1节），但是对全局声明无法做到这一点。C预处理程序可以通过宏处理引进根本无法预期的病态依赖性，从而可能造成很大的破坏（18.1节）。

我在某个时候曾经表述过自己关于避免微妙的解析方式的愿望，说：“帮助你下决心不应该是编译程序的事情”。换句话说，产生编译错误将比产生某种很含糊的解析更容易接受。对多重继承的带有歧义性的规则是这方面的一个好例子（12.2节）。关于重载函数的有歧义的规则是另一个例子，它也说明了在兼容性和灵活性的约束之下，要做好这件事有多么的困难（11.2.2节）。

如果有疑问，就选择有关特征的那种最容易教给人的形式：这是在不同可能性之中进行选择的第二位的规则，使用起来也很有技巧性，因为它可能变成一种关于逻辑美的争论，而且可能与熟悉不熟悉有关。写出描述它的辅导材料和参考手册，看看人们是否容易理解是这个规则的一种实践方式。这里的一个意图就是简化教学人员和维护人员的工作。还应该记住，程序员们并不笨，不应该通过付出重要的功能方面的代价去换取简单性。

语法是重要的（常以某些我们不希望的方式起作用）：保证类型系统的一致性，一般来说，保证语言的语义清晰、定义良好是最基本的东西。语法是第二位的，而且看起来人们能学会去喜爱任何语法。

当然，语法就是人们看到的东西，也是语言最基本的用户界面。人们喜爱某些形式的语法，并用某种奇特的狂热去表达他们的意见。我认为，想改变这些东西，或者不顾人们对某些特定语法的对抗情绪去引进新的语义或者设计思想，这些都是没有希望的。因此C++的语法，在设法使其更合理和规范的同时，也尽可能地避免去触犯程序员们的成见。我的目标是逐渐使一些讨厌的东西淡出，例如隐含的int（2.8.1节）和老风格的强制（14.3.1节）等等，同时又尽可能地减少使用更复杂形式的声明符语法（2.8.1节）。

我的经验是，人们往往过分热衷于通过关键词来引进新概念，以至于如果一个概念没有自己的关键词，教起来就非常困难。这种作用是很重要的、根深蒂固的，远远超过人们口头表述的对新关键词的反感。如果给他们一点时间去考虑并做出选择，人们无疑会选择新的关键词，而不是某种聪明的迂回方案。

我试着把重要操作做成容易看见的。例如，老风格强制的一个重要问题是它们几乎是不可见的。此外，我也喜欢把语义上丑陋的操作在语法上也弄成丑陋的，与语义相匹配，例如病态的类型强制（14.3.3节）。一般说也应该避免过分啰嗦。

应该清除使用预处理程序的必要性：如果没有C预处理程序，C语言本身和后来的C++可能早就是死胎了。没有Cpp它们根本就不能有足够的表达能力和灵活性，不能处理重要项目中所需要完成的任务。但在另一方面，Cpp丑陋低级的语义也使构造和使用更高级更优雅的C程序设计环境变得过分困难、代价过分昂贵。

因此，必须针对Cpp的每个基本特征找到能符合C++语法和语义的替代品。如果能够完成这个工作，我们就可能得到更便宜的大大改进了的C++程序设计环境。沿着这个方向，我们也将能除掉许多很难对付的错误的根源。模板（第15章）、在线函数（2.4.1节）、const（3.8节）和名字空间（第17章）都是在这个方向上留下的脚印。

4.5 低级程序设计支持规则

很自然，上面提出的规则基本上都适用于所有的语言特征。下面的规则同样也影响到C++如何作为一种表述高层设计的语言。

低级程序设计支持规则

- 使用传统的（笨[⊖]）连接程序
- 没有无故的与C的不兼容性
- 在C++下面不为更低级的语言留下空间（除汇编语言之外）
- 对不用的东西不需要付出代价（0开销规则）
- 遇到有疑问的地方就提供手工控制的手段

使用传统的（笨[⊖]）连接程序：最早的目标中就包括了容易移植，容易与用其他语言写的软件相互操作。强调C++应该可以用传统连接程序实现，也就是为了保证这些东西。要求通过在Fortran早期就有的连接技术必然是一件很痛苦的事情。C++的一些特征，特别是类型安全的连接（11.3节）和模板（第16章）都可以用传统的连接程序实现，但如果更多的连接支持，它们还可以实现得更好。C++的第二个目标就是想推动连接程序设计的改革。

采用传统连接程序，使得维持与C的连接兼容性变得相对比较简单了。对平滑地使用操作系统功能，使用C、Fortran等，使用库，以及写为其他语言所用的库代码，这种特性都是最本质性的。对于写想作为系统底层部分的代码，例如设备驱动程序等，使用传统连接程序也是最关键的事情。

没有无故的与C的不兼容性：C语言是有史以来最成功的系统程序设计语言。数以十万计的程序员熟悉C，现存的C代码数以十亿行计，存在着一个集中关注C语言的工具和服务产业。而且C++又是基于C的。这就带来一个问题：“C++的定义在与C相匹配方面到底应该靠得多么近？”C++不可能把与C的100%兼容作为目标，因为这将危及它在类型安全性和对设计的支持方面的目标。当然，在这些目标不会受到干扰的地方应该尽量避免不兼容性——即使这样做的结果不太优雅。在大部分情况下，已经接受的与C语言的不兼容性都是在那些C规则给类型系统留下大漏洞的地方。

在过去这些年里，C++最强的和最弱的地方都在于它与C的兼容性。这种情况并不奇怪。与C兼容性的高低将来也一直会是个重要议题。在今后的年代里，与C的兼容性将越来越少地被看作是一种优点，而更多地变成一种义务。必须找到一条发展的道路（第9章）。

在C++下面不为更低级的语言留下空间（除汇编语言之外）：如果一个语言的目标就是真正成为高级的——也就是说，它想完全保护自己的程序，使之避开基础计算机丑陋且使人厌倦的细节——那么它就必须把做系统程序设计的工作让给另外一些语言。在典型情况下这个语言是C。但也是很典型的，C在许多领域中将取代这种高级语言，只要在这里控制或者速度被认为是最关键的问题。常见的情况是，这将最终导致整个系统完全用C语言来编写；或者是导致了这样的一个系统，只有对两种语言都非常熟悉的人才能够把握它。在后一情况下程序员常常遇到一个艰难的选择：给定的任务究竟适合在哪个层次上做程序设计呢，他不得不同时记住两种语言的原语和准则。C++试图给出另一条路，它同时提供了低级特征和抽象机制，

[⊖] 笨（dumb）连接程序，指那些传统的最基本的，不提供任何特殊功能的连接程序，也是一般系统上所普遍使用的连接程序。——译者注

支持用这两种东西构造混合的系统。

为了继续成为一种可行的系统程序设计语言，C++必须保持C语言的那种直接访问硬件、控制数据结构布局的能力，保有那些能够以一对一的风格直接映射到硬件的基本操作和数据结构。这样它的替代品就只能是C或者汇编语言。语言设计的工作就是去隔离这些低级特征，使那些不直接对系统细节操作的代码就不需要这些低级特征。目标是保护程序员，防止出现不是有意越过界线的偶然的错误使用。

对不用的东西不需要付出代价（0开销规则）：对于规模大的语言，有一种人人皆知的论断，说它们会产生大而慢的结果代码。最常见的就是由于支持某些假设的高级特征而产生的开销，而这种开销又散布在整个语言的所有特征之中。例如，所有的对象都被扩大，以保存一些为某种系统内务操作而使用的信息；对于所有数据都采用间接访问方式，因为某些特征通过间接访问特别容易管理；或是对各种控制结构都进行了加工以迎合某种“高级的控制抽象”。这类“分布式的增肥”对C++而言是根本不合适的，接受它就会在C++之下为更低级的语言留下空间，使对于那些低级和高性能的应用而言，C语言成为比C++更好的选择。

这个规则在C++的设计决策中不断地成为最关键的考虑。虚函数（3.5节）、多重继承（12.4.2节）、运行时类型识别（14.2.2节）、异常处理和模板都是与此有关的特征实例，它们的设计部分地可以归于这条规则，都是到了我自己已经确信能够构造出遵守0开销原则的实现方式时，这些特征才被接受进来。当然，一个实现者可以决定在0开销规则和系统所需要的某些性质之间如何做一种折衷，但是也必须仔细地做。程序员通常对于分布式的增肥会有刺耳的情绪化的反应。

如果要拒绝人们建议的一个特征，0开销规则可能是所有规则中最锋利的一个。

遇到有疑问的地方就提供手工控制的手段：我对信任“高级技术”总是非常勉强，也特别不愿意去假定某些真正复杂的东西是普遍可用的和代价低廉的。在线函数是这方面的一个很好的例子（2.4.1节）。模板初始化是另一个例子，我在那里应该更当心一点，后来又不得不增加了一种显式控制机制（15.10节）。对于存储管理的细节控制也是一个例子，通过手工控制可能得到重要的收获，然而，只有时间才能告诉我们这些收获是不是可以通过某种自动化技术以类似的代价得到（10.7节）。

4.6 结束语

对于主要的语言特征，所有这些规则都是必须考虑的。忘掉任何一个就很可能带来某种不平衡，从而伤害到一部分用户。与此类似，让某个规则成为主导而损害其他方面也可能带来类似的问题。

我试着将这些规则都陈述为正面的命令式的句子，而不是构造起一个禁止表。这可以使它们从本质上说更不容易被用于排除新的思想。我对C++的观点是，它是一种生产软件的新语言，特别关注那些影响程序结构的机制，与只做一些小调整的自然倾向相比，这种观点走的完全不是同一条路。

ANSI/ISO委员会工作组对语言扩充提出了一个检查表（6.4.1节），那是对一个语言特征应该考虑的问题的更特殊、更细节的东西。

第5章 1985—1993年表

请注意，做事需要时间。

——Piet Hein

Release 1.0之后的年表——Release 2.0——2.0特征概览——《带标注的C++参考手册》(*The Annotated C++ Reference Manual, ARM*)和非官方的标准化——ARM特征概览——ANSI和ISO标准化——标准特征概览

5.1 引言

第二部分将描述为完成C++而增加的特征，那里的描述是围绕着语言特征组织起来的，而不是按照年表顺序。在本章里给出有关的年表。

不按照年表组织描述，是因为实际时间顺序对于C++的最后定义已经不那么重要了。我知道在一般情况下语言将向什么方向发展，需要着手处理什么问题，而为了处理它们可能需要使用哪类特征。当然，这并不是说我就可以简单地坐下来，直接去完成语言的某一个重要修订。这样可能会花太长的时间，而且会使我在某种真空中工作，得不到最重要的反馈信息。因此语言的扩充是逐渐发展的，一点点加进语言里的。实际的顺序对于那个时代的用户当然很重要，对于在所有时间里保持语言的一致性也是极端重要的。当然，对于C++最后的形态而言，这些东西就不那么重要了。按照年代顺序展示这些扩充有可能使人很难理解语言的逻辑结构。

本章将讨论导致Cfront的Release 2.0的工作，导致《带标注的C++参考手册》的工作，以及在标准化方面的努力：

1986—1989 Release 2.0给C++填充了诸如抽象类、类型安全的连接、多重继承等特征，但是没有增加任何真正新的东西。

1988—1990 《带标注的C++参考手册》增加了模板和异常处理，这样做实际上给实现者提出了一个重要的新挑战，也为在怎样写C++程序方面的剧烈变化打开了一条通路。

1989—1993 标准化的努力为C++程序员的工具箱增加了名字空间、运行时的类型识别，以及许多小的新特征。

在所有这三个阶段里我都做了许多工作，设法把C++的定义弄得更准确，通过一些小改变来清理这个语言。按照我自己的看法，这是一种持续不断的努力。

5.2 Release 2.0

到了1986年中期，对于所有关心C++的人而言，这个语言已经踏上了征途：关键的设计决策都已经做出，未来发展的方向也已经设定，目标是参数化类型、多重继承和异常处理。

还需要基于实验取得更多经验并做出调整，但是赞美荣耀的日子已经过去了。C++从来也没有愚蠢的遗憾，但是在当时已经再也没有做剧烈改变的现实可能性了。无论是好是坏，做过的事情都已经做了，剩下的就是无穷无尽的实实在在的工作。在那个时刻全世界大约有2000个C++用户。

正是在这个时刻有了一个计划——最早是Steve Johnson 和我接到的——由一个开发和支持组织承担起在工具（最重要的就是Cfront）上的日常工作，以使我能有时间去做新特征和预计将来要依靠它们的库方面的工作。也是在这时，我开始预期首先是AT&T，而后是其他机构将开始构造编译系统和其他工具，这些东西最后将使Cfront变成过时之物。

实际上他们早就开始了。但是很好的计划总是很快就出了轨，因为开发管理部门的犹豫不决、不称职和缺乏注意力焦点等。一个开发全新C++编译系统的项目从Cfront的维护开发中转走了注意力和资源。一个在1988年早期发布Release 1.3的计划由于各种瑕疵而完全失败。这件事的影响是我们必须为Release 2.0等到1989年6月。虽然Release 2.0几乎在所有方面都比Release 1.2好得多，但它也没能提供在“什么是”论文（3.15节）中给出梗概的那些特征，而且——部分地也是受到影响——它没有带上一个经过重大改进的范围广泛的库。发布一个这样的库是可能的，因为其中许多东西在那时已成为USL[⊖] 的标准部件库，已经在AT&T作为内部产品使用了一段时间。当然，对于直接支持模板的希望仍然遮住了我的眼睛，使我没有看到其他替代方式。当时在某些开发管理者中存在着一种不切实际的信念，那就是说库可以既成为一项标准，又是一个重要的收入来源。

Release 2.0是由Andrew Koenig，Barbara Moo，Stan Lippman，Pat Philip和我这一个组完成的工作。Barbara协调，Pat集成，Stan和我编码，Andy和我评价错误报告并讨论语言的细节，Andy和Barbara做测试。总而言之，我为2.0实现了所有的新特征并更正了大约80%的错误，此外我还写出了大部分文档。像过去一样，语言设计问题和参考手册的维护都是我的责任。Barbara Moo和Stan Lippman已经逐渐成为这个队伍的核心，他们后来做出Release 2.1和3.0。

许多影响了带类的C和原来的C++的人仍然以不同的方式为这个进步提供了帮助。在[Stroustrup, 1989b]中特别感谢了Phil Brown，Tom Cargill，Jim Coplien，Steve Dewhurst，Keith Gorlen，Laura Eaves，Bob Kelley，Brian Kernighan，Andy Koenig，Archie Lachner，Stan Lippman，Larry Mayka，Doug McIlroy，Pat Philip，Dave Prosser，Peggy Quinn，Roger Scott，Jerry Schwarz，Jonathan Shapiro以及Kathy Stark。在这一阶段的语言讨论中，最活跃的人物是Doug McIlroy，Andy Koenig，Jonathan Shapiro和我。

语言及其实现的稳固性被认为是最基本的要求 [Stroustrup, 1987c]：

“应该强调指出，这些语言修改都是扩充，C++过去是、将来也还是一个为长周期的软件开发所使用的稳固的语言。”

这也是C++作为工业使用的通用语言所应该扮演的角色 [Stroustrup, 1987c]：

“至少有一些C++实现具有可移植性，这是一个关键性的设计目标。由此出发，可能在移植中花费大量时间或者对C++编译系统提出太强的资源要求的扩充都被拒绝了。语言发展的

[⊖] USL是那时AT&T中支持和发布UNIX及相关工具的一个组织，后变成一个单独公司，称为UNIX系统实验室；再后来被Novell买下。

这种思想是与另外一种可能发展方向截然不同的，那种方向就是为了程序设计的方便而：

- 付出效率或者结构方面的代价；
- 为了新手考虑而放弃通用性；
- 为在某个特定领域中使用而给语言增加特殊用途的特征；
- 为增进与某种特定的C++环境的集成性而增加语言特征。”

Release 2.0是个大进步，但它并没有提供任何真正的新东西。那时我愿意这样解释说“所有2.0的特征——包括多重继承——都是为了去掉一些约束，我们已经认为它们是过于严格了，所以就去掉了它们。”这实际上是一种夸大其词，不过是以一种世故的方式反对人们对每个新特征的高估倾向罢了。从语言设计的观点看，Release 2.0最重要的方面在于它增强了一些语言特征的普遍性，改进了它们与整个语言的集成性。从一个用户的观点看，我认为Release 2.0最重要的方面是更稳定的实现和大大改善了的支持。

特征概览

2.0的主要特征最早是在 [Stroustrup, 1987c] 中给出的，后来在这篇文章的修订版本 [Stroustrup, 1989b] 中做了总结，它作为文档的一部分与2.0相伴：

- 1) 多重继承 (12.1节)
- 2) 类型安全的连接 (11.3节)
- 3) 对重载函数的更好解析方式 (11.2节)
- 4) 赋值和初始化的递归定义 (11.4.4节)
- 5) 为用户定义存储管理而用的更好的机制 (10.2节, 10.4节)
- 6) 抽象类 (13.2节)
- 7) 静态成员函数 (13.4节)
- 8) `const` 成员函数 (13.3节)
- 9) `protected` 成员 (最先是在Release 1.2中提供) (13.9节)
- 10) 推广了的初始化机制 (3.11.4节)
- 11) 基和成员的初始化描述机制 (12.9节)
- 12) 对运算符`->`的重载 (11.5.4节)
- 13) 到成员的指针 (13.11节)

这些扩充和精化中的大部分表现的都是由C++取得的经验，由于我过去没有更多的深谋远虑，所以它们根本不可能被更早加进来。很自然，这些特征的集成涉及到大量的工作，但其中最不幸的是这些工作被允许取得了优先权，超过了完成在“什么是”文章 (3.15节) 中给出梗概的那个语言的工作。

大部分特征都以这种或者那种方式提高了语言的安全性。Cfront 2.0完成了跨过编译单位的边界的对函数类型的一致性检查，使重载的解析与顺序无关，并且能够确定更多的有歧义性的调用。`const` 概念变得更容易理解了，指向成员的指针堵住了类型系统中的一个漏洞，这里提供的显式地针对类的专用分配和释放操作，使得原来“通过给`this`指针赋值”这种易出错的技术 (3.9节) 完全过了时。

在所有这些特征中，1)、3)、4)、5)、9)、10)、11)、12) 和 13) 是我在1987年USENIX会

议上展示（7.1.2节）时就已经在AT&T内部使用的东西。

5.3 《带标注的C++参考手册》

在1988年的某个时候，事情已经变得清楚起来，C++语言最终将被标准化 [Stroustrup, 1989]。这时已经产生了少数几个独立的实现。很清楚，必须努力去写出一个更精确更完整的语言定义，进一步说还必须让这个定义得到广泛的认可。在开始时，正式的标准化还没有被当作一种考虑，许多涉足C++的人当时认为——至今仍然认为——在取得真正的经验之前做标准化是件可恶的事情。当然，要做出一个改进的参考手册可不是那种某一个人（我）自己就可以完成的事情，需要从C++社团来的输入和反馈。这样我就开始有了这个想法，重新写C++的参考手册，并在世界范围C++社团中重要的有见识的成员中传播它的草稿。

大约在同一个时候，作为AT&T销售C++的部门（USL）也希望有一个新的改进了的C++参考手册，并把写这个手册的任务交给了它的一个雇员，Margaret Ellis。看起来惟一合理的选择是合并这种努力，产生一个经过广泛审阅的参考手册。在我看来事情也很明显，如果在出版这个手册时带上一些附加信息，将能有助于对这个新定义的广泛接受，也能使C++得到更广泛的理解。这样最后写的就是《带标注的C++参考手册》(*The Annotated C++ Reference Manual*) [ARM]：

“为C++的未来发展提供一个坚实的基础……并且能成为C++正式标准化的一个出发点。……这个C++参考手册独立地提供了C++语言的一个完整定义，但是这里简练的参考手册风格也留下了许多没有回答的问题。关于什么不在这个语言里，为什么某个特征定义成当前这个样子，以及人们可以如何实现某个特定特征，等等，这些都不应该出现在参考手册里。但无论如何，这些东西也是大部分用户感兴趣的。这些讨论将以标注的方式表达，写在另外的注释节里。

这些注释也能帮助读者重视语言中不同部分之间的关系，一些需要强调的观点和一些表述之外的东西，在参考手册里这些东西常常被人们忽视。例子和与C语言的比较也使这本书比干巴巴的参考手册更容易被接受。”

经过与生产部门的人们的一些小口角，最后达成了统一意见：我们应该把ARM（人们通常这样称《带标注的C++参考手册》）写成描述了完整的C++语言，也就是说包括模板和异常处理，而不是作为由当时最新的AT&T发布所实现的子集的手册。这也是非常重要的，因为这将要以一种与C++的所有实现都不同的方式，清楚地建立起这个语言本身。这个原则在刚开始就提出来了，但是还需要反复地说，因为似乎用户、实现者和销售人员都很难记住这件事情。

关于ARM，我写了整个参考手册里的每个词，除了关于预处理器（第18章）的一节，那是Margaret从ANSI C标准中移植过来的。有关标注和注释的节是合作写出的，其中一些部分是基于我过去的一些文章 [Stroustrup, 1984b, 1987, 1988, 1988b, 1989b]。

ARM中原本意义上的参考手册经过了来自20多个组织的大约一百人的审阅，大部分名字都已经列入ARM的致谢一节里，他们也对整个ARM做出了很大贡献。特别值得提出的是Brian Kernighan、Andrew Koenig和Doug McIlroy的贡献。ARM中的原本意义上的参考手册在1990年3月被接受作为ANSI C++标准化的基础。

ARM并没有解释这个语言的特征所支持的技术：“这本书不企图去教C++的程序设计，它只解释这个语言是什么，而不是如何使用它 [ARM]。”有关的工作留给了《C++程序设计语言》的第2版 [2nd]。不幸的是，一些人并没有重视这个声明，结果常常造成一种观点，认为C++不过是一些模糊细节的汇集；另外也导致人们写不出优雅的易维护的C++代码，见7.2节。

特征概览

ARM里描述的一些不太重要的新特征直到AT&T的Release 2.1和其他提供商的C++编译系统中才得以实现。这其中最明显的就是嵌套的类。来自参考手册外部审阅者的许多意见都督促我恢复嵌套类作用域原来的定义。我也早已对让C++的作用域规则在有C规则的地方与之保持一致感到失望（2.8.1节）。

ARM里提出的最重要的新特征是模板（第15章）和异常处理（第16章）。此外，ARM还允许对增量（++）的前缀和后缀形式分别进行重载（11.5.3节）。

为了与ANSI C保持一致，这里也允许局部静态数组的初始化。

为与ANSI C一致而引进了`volatile`修饰符，以帮助优化程序的实现者们。我完全不能保证它与`const`在语法形式上的平行性是否意味着语义上的类似。无论如何，我对`volatile`并没有很强的感觉，也看不到有什么理由要在这个领域中试着去改进ANSI C委员会的决定。

总结一下，ARM提出的特征是：

- 2.0的特征（5.2.1节）
- 模板（第15章）
- 异常（第16章）
- 嵌套的类（13.5节）
- 对++和--的前缀和后缀形式分别进行重载（11.5.3节）
- `volatile`
- 局部静态数组

ARM特征（除了异常处理）直到1991年9月Cfront的Release 3.0中才成为广泛可用的东西。所有ARM的特征集合于1992年初在DEC和IBM的C++编译系统中第一次可用。

5.4 ANSI和ISO标准化

自1990年以后，ANSI/ISO的C++标准化委员会就变成了为完成C++的努力的最重要的论坛。

进行正式（ANSI）C++标准化的动议是由Hewlett-Packard提出的，附议的有AT&T、DEC和IBM。来自Hewlett-Packard的Larry Rosler在这个动议中起了重要作用。特别是在1988年末的某个时候Larry来找我，我们讨论了关于需要做正式标准化的问题。关键问题是什么时间，Larry站在大用户的角度希望能尽快，我则表示希望推迟，以便能在标准化之前做更多的试验，以取得更多的经验。在权衡了许多尚不明朗的技术与商业考虑之后，我们达成了一致意见，在开始官方标准化之前还应该有大约一年的时间，以便我们有更多的机会获得成功。按照我的记忆，ANSI C委员会的第一次技术会议正好是在我们留出的一年到期之前3天召开的（1990年3月）。

关于ANSI标准化的建议书是Hewlett-Packard的Dmitry Lenkov写的 [Lenkov, 1989]，Dmitry的建议列举了立即进行C++标准化的一些理由：

- 与大部分其他语言相比，C++正在经历一个更快的公众接受过程。
- 延迟……将导致许多方言。
- 需要为C++提供严密的细节化的完整语义定义……对每个语言特征。
- C++缺乏某些重要特征……包括异常处理，多重继承方面的东西，支持参数化多态性的特征，以及标准库。

这个建议也特别强调了与ANSI C兼容的必要性。ANSI C++委员会X3J16的组织会议于1989年12月在华盛顿特区召开，出席者大约有40人，包括参加C++标准化的人，那些被认为是“老C++程序员”的人，以及一些其他人。Dmitry Lenkov成为委员会的主席，Jonathan Shapiro是它的编辑。

第一次技术会议由AT&T做东，1990年3月在新泽西的Somerset召开。AT&T得到了这个荣誉并不是因为任何有关这个公司在对C++的贡献方面的评判，而是因为我们（出席华盛顿会议的X3J16成员）决定根据天气情况安排第一年的会议。这样，Microsoft公司7月在西雅图主办了第二次会议，Hewlett-Packard公司11月在Polo Alto主办了第三次会议。按照这种方式我们在所有这三次会议期间都遇到绝好的天气，也避免了公司之间很容易表现出的位置之争。

这个委员会现在有超过250个成员，开过大约70次会议。委员会的初始目标是在1993年末或者1994年初写出标准的草稿，供公开审议，希望在大约两年之后完成正式标准。对于一个程序设计语言的标准化而言，这是一个非常含糊的安排。比较一下，C语言的标准化用了7年时间。当前的安排是要求在1995年4月给出供公众审议的标准草稿，我想我们很有可能按时完成^⑨。

当然，C++的标准化并不只是美国关心的事情，从一开始，就有来自其他国家的代表参加ANSI C++会议。1991年6月在瑞典的Lund召集了ISO C++委员会WG21，两个C++标准委员会决定立即在Lund召开联合会议。来自加拿大、丹麦、法国、日本、瑞典、英国和美国的代表出席了会议。特别值得提出的是，这些来自不同国家的代表中的绝大部分都实际上是很长时间的C++程序员。

C++委员会有一个很困难的特许状：

- 语言的定义必须准确而全面。
- 考虑C/C++的兼容性。
- 必须考虑超出当时C++实践的各种扩充。
- 必须考虑库。

在所有这些之上，C++社团的人们之间的差异已经非常大，而且完全无组织，因此标准委员会自然就成为这个社团的主要注目点。从短期来看，委员会最重要的角色实际上是：

“C++委员会是一个场所，在这里写编译系统的人、写工具的人、他们的朋友和代表可以聚在一起，讨论语言的定义以及实现（在商业竞争允许的范围内）问题。这样C++委员会就已经为C++社团提供了服务，因为它帮助把各种实现相互间变得更加类似（更“正确”），采用的

^⑨ 草案如期发布供公开评议。经过第二轮评议后，这个标准草案进入表决程序。它被一致通过。ISO C++标准在1998年11月被所有参与国批准，成为一个国际标准。

方式是提供一个论坛来公开讨论有关的问题。如果没有它，在发现了ARM里没有回答的问题时，写编译程序的人就只能自己或者与几个朋友一起做某种猜测。或许他们会发个邮件给我——或许不——况且我也不可能处理每个问题。而这些问题一定会出现，某些人确实觉得以个人的方式处理它们不是一种正确的方式。没有委员会，就将不可避免地导致各种方言，委员会能够抑制这种倾向。我不能设想某个或某些人如果不直接或间接地与委员会有关系，现在还有可能做出一个工具，使它能成为与C++市场的主要活动厂商所形成的共识相一致的那一类东西[Stroustrup, 1992b]。”

标准化绝不是一件容易的事情。在委员会里有各种各样的人。有的人来这里就是为了维持现状；有的人带着一个有关现状的想法，希望能够把时间拨回到几年之前；有的人希望能与过去做彻底的决裂，设计出一个全新的语言；有的人只关心某一个问题；有的人只关心某一类系统；有的人在投票时完全按照他们雇主的脸色行事；有的人只代表他们自己；有的人带着有关程序设计和程序设计语言的理论观点；也有的人希望的是今天就有一个标准，即使这意味着遗留下许多没有结论的问题；有的人则除了一个完美的定义之外什么也不能接受；有的人还认为C++完全是一个新的语言，几乎没有用户；有的人则代表着在过去的十年里写出了成百万行代码的用户；如此等等。在标准化的规则下，我们都必须或多或少表示同意。我们必须达成“一致”（通常定义为一个很大的多数）。存在一些合理的规则——即使不太合理，它们也是委员会必须遵守的国家或者国际的规则。所有的利益都是合法的，让一个多数压服一个很大的少数的利益，最终将会产生一个标准，它只对某个过分狭隘的用户社团有用处。这样，委员会的每个成员都需要学会尊重那些看起来是异己的观点，学会妥协。这些倒是很符合C++的精神。

C兼容性是我们必须面对的第一个争论最多的问题。经过一些偶然出现的激烈争辩之后，才确定下100%的C/C++兼容性并不是一种选择。在这里也没有低估C兼容性的重要性。C++是个独立的语言，它并不是ANSI C的一个严格超集，也不可能被修改成一个这样的超集，又不严重地削弱由C++类型系统所提供的保证——而且也不打破数以百万行计的已有的C++代码。与此类似，任何显著削弱与C的兼容性的行为也将打破已有的代码，并使构造和维护混合的C和C++系统，以及从C到C++的转变都变得更加困难。这个决定常常被说成是“尽可能地接近C——但又不过分接近”，这是在Andrew和我写的一篇同名文章之后[Andrew, 1989]。思考C++及其发展方向的个人和小组已经一次又一次地得到同样的看法。在C++和ANSI C各自独立地对原来的C手册做了一些修改之后，如何达到“尽可能地接近C——但又不过分接近”就成为标准化委员会工作中一个主要的部分。Thomas Plum在这个工作中做出了重要贡献。

特征概览

在1994年11月Valley Forge会议之后由标准委员会给出的工作文件里提出的C++特征可以总结如下：

- 在ARM中刻画的特征（5.3节）
- C++的欧洲字符集表示（6.5.3节）
- 对于重载函数返回类型放松规则（13.7节）
- 运行时类型识别（14.2节）
- 条件语句中的声明（3.11.5节）

- 在枚举基础上的重载 (11.7.1节)
- 用户定义的对数组的分配和释放运算符 (10.3节)
- 嵌套类的预先声明 (13.5节)
- 名字空间 (第17章)
- 变化量 (13.3.3节)
- 新的强制 (14.3节)
- 布尔类型 (11.7.2节)
- 显式模板实例化 (15.10.4节)
- 在模板函数调用里的显式模板参数描述 (15.6.2节)
- 成员模板 (15.9.3节)
- 类模板作为模板的参数 (15.3.1节)
- 整类型的const static成员可以在类声明里用常量表达式进行初始化。
- 显式的建构函数 (3.6.1节)
- 异常描述的静态检查 (16.9节)

详情见6.4.2节。

第6章 标 准 化

你别想骗我，
我在早餐麦片粥里
放的奇怪东西比你可多得多了。
—— Zaphod Beeblebrox

什么是标准？——C++标准工作的目标——委员会如何运作？——委员会有哪些人？——语言净化——名字查找规则——临时量的生存期——语言扩充的准则——扩充建议列举——关键词实参——指数运算符——受限指针——字符集

6.1 什么 是 标 准

在程序员的心目中，对于标准是什么以及它应该是什么常有许多疑惑。关于标准的一种想法是，它应该能完全地描述什么样的程序是合法的，并准确地说明每个合法程序的意义。至少对于C和C++而言情况并不是这样。实际上，对于要设计语言去开拓硬件体系结构和有关装置这样一个缤纷的世界而言，上面的提法不是也不应该是一种理想。对于这样的语言，保持其行为相对于实现的独立性是一种最根本的东西。这样，标准就常常被说成是“在程序员和语言实现者之间的一个协议”，它不仅要描述什么是“合法的”源代码正文，还要一般性地说明程序员可以依靠什么，哪些东西将是与具体实现有关的。例如，在C和C++里可以将一个变量声明为int类型，但是标准就没有明确说明int到底有多大，只说明它至少应该有16位。

标准实际上应该是什么，最好用哪些术语去表述它，关于这些都可以有很长的带学究味道的争论。当然，最关键的问题还是对什么是、什么不是合法的程序做出明确的划分，并进一步明确规定，在所有的实现中哪些行为应该都一致，哪些应该依赖于具体的实现。怎样精确地做出这些划分是很重要的，但实际程序员对这些没多大兴趣。委员会的大部分成员特别关注的是标准中更具语言技术性的方面，标准到底应该标准化哪些东西的问题也很棘手，处理这个问题的重担主要就落在委员会的项目编辑的肩上了。幸运的是，我们原来的项目编辑Jonathan Shapiro对这类事情很有兴趣。Jonathan现在已经从这个位置上退了下来，把工作交给了Andrew Koenig，但他仍然是委员会的成员。

另一个有趣的（或者说是困难的）问题是，在一个实现中带有多少个标准里未描述的特征还能够被接受。禁止所有这类扩充当然不合理，毕竟对于C++社团中一些非常重要的派系而言，某些扩充可能是必需的。例如某些机器带有支持某种特殊并行机制的、或者特殊寻址约束的硬件，或者特别的向量硬件等。我们不能让每个C++用户为支持所有这些互不兼容的专用扩充而烦恼，因为它们是不兼容的，通常也会给不用它们的用户增加负担。但阻止实现者去为这部分社团服务也是不合适的，只要他们在自己的基本扩充之外能与大家步调一致。在另一方面，有一次我也看到过一个“扩充”，其中居然允许从程序的任何函数里访问类的私用成员，也就是说实现者根本就不管访问控制问题。我不会认为这是个合理的扩充。斟酌标

准的字句，允许前一种东西并排斥后一种东西可绝不是一件简单的事情。

这里有一个重点，就是要保证非标准的扩充是可检查的，否则就可能出现程序员在某个早晨起来，发现某些重要代码依赖于供货者的独特扩充，这样就无法在合理的方便的范围内考虑改变供货者的可能性了。记得在我还是一个大学新生时，曾惊喜地发现在我们大学主机上的Fortran是一种“扩充的Fortran”，带有许多非常好的特征。而当我后来认识到，这实际上意味着我的程序除了在CDC6000系列计算机上可用之外就毫无用处之后，惊喜最终变成了一种沮丧。

这样，对C++而言，与标准相一致的程序的100%可移植性一般来说是不能做到的，也不是一种理想。符合标准的程序不能100%可移植，是因为它可能显示出某些与实现有关的行为。实际上它必然会这样。例如，如果一个完全合法的C或者C++程序可能恰好依赖于将内部求余数运算符%作用到负数上的结果，在不同情况下它的意义就会改变。

进一步看，实际程序大都依赖于某些提供服务的库，而这些未必是每个系统都提供的。例如，一个Microsoft Windows程序不经修改大概不能在X[⊕]下运行，要把一个使用了Borland基本类的程序移植到能够在MacApp[⊖]下运行也绝不是一项简单工作。实际程序的可移植性要靠设计，需要把依赖于实现和环境的东西封装起来，而不可能仅仅是源于它符合标准文档中的某几条规则。

理解一个标准不能保证哪些东西，与理解它到底允诺了些什么至少是具有同等的重要性。

6.1.1 实现细节

看起来每周都有新的请求，提出一些需要标准化的东西，例如虚表的布局，在类型安全的连接中名字的编码模式，甚至是排错程序等。当然，这些都是与实现特质有关的问题，或者是实现的细节，已经超出了标准的范围。用户当然可能希望用某个编译系统编译出来的库能够与另一个编译系统编译的代码一起工作，可能希望二进制代码能从一种体系结构的机器搬到另一种机器，可能希望排错系统能够独立于需要用它去检查的代码，与编译程序的实现无关。

很自然，指令集合、操作系统接口、排错系统所使用的格式、调用序列、以及对象布局形式的标准化等等，都已远远超出了一个程序设计语言标准化组织的能力范围，因为一种程序语言只不过是一个大得多的系统中的一支棒。这种最普遍的标准化甚至可能并不是人们所期望的，因为它会窒息机器系统结构和操作系统的进步。如果一个用户真需要完全地独立于硬件、系统或环境，那就必须去构造一个解释器，让它带着自己为应用所提供的标准环境。这种方式也有其本身的问题，特别是它们很难利用特殊硬件的能力，无法遵循局部的方式准则。如果通过与某种允许不可移植性的语言接口的方式来克服上述问题，例如与C++接口，那么所有的问题又重新出现了。

对任何适用于严肃的系统工作的语言，我们都必须接受这样的事实：每个今天或者将来的新用户都可能在网络上发一条消息：“我把我的目标代码从我的Mac搬到了SPARC上，它就不干活了。”与可移植性类似，互操作性也是设计问题，是理解由环境所强加的限制的

⊕ 指UNIX下的X Window系统。——译者注

⊖ Apple公司Mac机器中的应用程序支撑功能。——译者注

问题。我常常遇到这样的C程序员，他们并没有意识到在同一个系统上的两个不同C编译生成的代码并不能保证能够连接到一起，事实上这通常都是行不通的——然而他们却表达出一种厌恶，说C++不能保证互操作性。和许多情况下一样，我们的一个重要工作就是教育用户。

6.1.2 现实的检查

对于一个标准化委员会，除了许多形式上的约束之外，还有许多非形式的和来自实践的约束：许多标准根本就不为它们所预期的用户所注意。例如，Pascal和Pascal 2标准几乎完全被遗忘了。对于大部分Pascal程序员而言，“Pascal”实际意味着Borland公司的那个做了极大扩充的Pascal方言。Pascal标准没有提供用户认为最关键的许多特征，而Pascal 2标准一直不能出台，直到另一个不同的非形式的“工业标准”建立起自己的位置。另一个具有警示性的例子是，在UNIX上大部分工作仍然是在K&R C上做的，ANSI C还正在这个社会里奋斗。个中原由，看起来是由于某些用户没看到ANSI/ISO C与K&R C比较上的技术优越性，并过高估计了转变的短期成本。甚至一个不那么富有挑战性的标准也只能慢慢找到它的应用之路。为了能够被公众接受，一个标准必须适时地反映用户们的需求。在我看来，要想为一个好语言发布一个好标准，采用某种适时的方式是非常关键的。试图把C++转变成一种“完美的”语言，或者希望能够做出一个任何人都不可能误读的标准——无论他如何偏执或者缺乏教育——这也远远超出了委员会的能力（3.13节）。事实上，这超出在一个大用户社团和时间的约束下任何人的工作能力（7.1节）。

6.2 委员会如何运作

实际上存在着若干个为标准化C++而成立的委员会。第一个，也是最大的，就是美国国家标准局（ANSI）的ANSI-X3J16委员会。这个委员会对计算机和商用设备生产联盟（Computer and Business Equipment Manufacturers Association, CBEMA）负责，并在它的规则之下运作。特别是这将意味着一个公司一票的投票制度，没在公司工作的个人也作为一个公司来看待。任何成员在其参加的第二次会议上就可以开始投票。从官方角度看，最重要的委员会是国际标准化组织（ISO）的ISO WG-21。这个委员会在国际的规则之下运作，而且将最后做出一个国际标准。特别是这意味着一个国家一票的投票制度。其他一些国家，包括英国、丹麦、法国、德国、日本、俄罗斯和瑞典等，现在也有了它们自己国家的有关C++标准化的委员会。这些国家的委员会也给ANSI/ISO联合会议寄送请求信、推荐书，并派代表来参加会议。

我们决定基本上不接受任何不能同时在ANSI和ISO两个投票中都通过的东西。这意味着委员会的运作很像一个两院制的国会，其“下院”（ANSI）完成大部分争论，而“上院”（ISO）认可下院提出的决定，只要它们有意义并充分地反映了国际社会的利益。

在一个偶然的情况下按这个程序工作拒绝了一个建议，如果不是这样的话，它就会因为一个很小的多数而被通过。我认为各国代表使我们避免了一次犯错误，而这个错误又可能造成更大的不和。我不能解释说大多数就反映了统一意见，因此我认识到——与这个建议的技术价值并没有关系——来自各国的代表给了委员会一个重要的提醒，针对在他们的特许之下

委员会的责任问题。当时讨论的问题是：C++是否应该对定义最小翻译规模的限制有一个特别的说法。一个经过重大改进后的建议在后来的会议中被接受了。

ANSI和ISO委员会每年在一起开三次会。为了避免混乱，我在下面将用一个委员会来称呼它们。每次会期为一周，其中的许多小时花在受合法委托的程序性事务上，然而更多的时间则被花费在某种混乱上，你可以想象当70个人试着去理解一个问题到底是什么时会出现怎样的情况。白天的有些时间和几个晚上作为技术性讨论时段，在这里提出和讨论重要的C++论题，如国际化字符的处理和运行时类型识别；以及另一些与标准工作有关的问题，如形式化方法和国际标准的正文组织等。其他时间主要用于工作组会议，有关讨论都是基于工作组的报告进行的。

当前的工作组有：

- C兼容性
- 核心语言
- 编辑
- 环境
- 扩充
- 国际化问题
- 库
- 语法

很清楚，在一年里只有三周的会期中，需要处理的工作实在太多了，因此大部分实际工作都是在会议之间做的。为了帮助交流，我们大量地使用了email。每次会议都涉及大约三英寸厚双面印刷的备忘录。这些备忘录以两个包裹的形式邮寄：一个在开会之前几个星期到达，以帮助成员们做好准备；另一个在会后几个星期，反映从前一个包裹到会议结束这一段时间所完成工作的情况。

C++标准化委员会里有哪些人

C++委员会由在利益、关注点、背景等方面差别极大的一些个人组成。有些人代表的就是他们自己，有些则代表着庞大的公司。有些使用PC，有些使用UNIX，有些使用大型机等等。有些人使用C++，也有些并不使用。有些人希望C++成为一个更加面向对象的语言（又依照许多不同的有关“面向对象”的定义），另一些人如果看到ANSI C成为C语言发展的终点会觉得更舒服些。许多人有着C语言的背景，也有些没有。有的人有标准化工作的背景，大部分人没有。有些人有计算机科学的背景，有些人没有。有些人是程序员，有些人不是。有些人是最终用户，有些人是工具的提供者。有些人对大项目感兴趣，也有些人不感兴趣。有些人对与C的兼容性感兴趣，有些人则没兴趣。

除了按照正式说法所有的人都不付酬的志愿者（虽然大部分人代表公司）之外，很难再找到这些人之间有什么共同点。这是很好的，只有一个丰富多彩的组织才能保证C++社团中丰富多彩的利益都能够得到代表。但这也确实常常使建设性的讨论特别困难，进展很缓慢。特别是这种非常开放的进程很脆弱，时常被一些个人所打断，由于他们的技术或者个人层次还没有成熟到能理解和尊重其他人意见的程度。我也担心C++用户（C++应用程序员和设计师们）的声音可能被语言专家们、所谓的语言设计师们、标准化官员们以及实现者们的声音所

淹没。

通常有大约70人出席会议，其中大约有一半人出席了每一次会议。投票人和代理人、观察员成员的数目超过250人。我是一个代理人成员，这个说法的意思是我代表着我的公司，但是另外有人为我的公司投票。让我给你一个印象，看看谁在这里有代表，简单地浏览一下1990年的成员表，列出一些大家都知道的名字：Amdahl, Apple, AT&T, Bellcore, Borland, British Aerospace, CDC, Data General, DEC, Fujitsu, Hewlett-Packard, IBM, Los Alamos National Labs, Lucid, Mentor Graphics, Microsoft, MIPS, NEC, NIH, Object Design, Ontologics, Prime Computer, SAS Institute, Siemens Nixdorf, Silicon Graphics, Sun, Tandem Computers, Tektronix, Texas Instruments, Unisys, US WEST, Wang, 以及Zortech。这个表当然是偏向于我所知道的公司和大的公司，但我也希望你能看到工业界确实得到了很好的代表。自然，所涉及的个人与他们所代表的公司同样重要，但我忍住了，不想把这个表变成了一个提出我的朋友名字的广告。

6.3 净化

最好的标准化工作中的大部分内容对普通程序员而言是看不到的，看起来也相当深奥，描述出来一般很枯燥。这是因为大量的工夫被花在如何找出一些方式，能够清楚完全地描述“每个人都知道，但是恰好没有在手册里说出来”的东西；还花在消解一些模糊点上，而这些东西——至少在理论上——对于大多数程序员不会有影响。当然，对于实现者而言，如果想保证能正确处理语言所有的给定使用，这些问题就非常重了。转过来说，这些问题对于程序员也是最关键的，因为即使是以最严密的方式写出的大程序也可能有意地或者偶然地依赖于某些特征，而它们中的某些正好是模糊的或者是深奥的。除非实现者们都赞成同一个意见，否则程序员在不同的实现和变成某个个别的编译办管理员的人质之间就没有多少选择了——这也与我对C++应该是什么（见2.1节）的观点相矛盾。

我将描述两个问题，名字查找问题和临时量的生存期问题，以说明其中的困难和所完成的工作细节。委员会把大部分工夫花在这类问题上。

6.3.1 查找问题

在C++定义里最难驾驭的问题常与名字的查找有关：一个名字的使用究竟引用的是哪个声明？在这里我将只描述一类查找问题：与类成员声明的顺序依赖性有关的问题。考虑：

```
int x;

class X {
    int f() { return x; }
    int x;
};
```

在这里`X::f()`引用的是哪个`x`？还有：

```
typedef char* T;

class Y {
    T f() { T a = 0; return a; }
```

```
typedef int T;
};
```

在这里Y::f()使用的是哪个T?

ARM给出了答案：在X::f()里的x引用的是X::x，而类Y的定义则是个错误，因为类型T在Y::f()里被使用之后它的意义又改变了。

Andrew Koenig, Scott Turner, Tom Pennello, Bill Gibbons和另外几个人在连续的几次会议上，花了许多时间去寻找对这类问题的精确的、完全的、有用的、符合逻辑的，而且是兼容的（与C语言标准和现存的C++代码兼容）的回答，在会议之间也花了若干个星期的时间。我对这些讨论的涉足程度有限，因为需要集中精力考虑扩充问题。

困难来自不同目标之间的相互冲突：

- 1) 我们希望只读一遍源代码正文就能完成语法分析。
- 2) 对类成员做记录不应该改变类的意义。
- 3) 将一个成员函数的体以显式的在线形式写出来，应该与以不在线的形式写出具有同样的意义。
- 4) 来自外层作用域的名字应该能在内层作用域里使用（按照它们在C里同样的方式）。
- 5) 名字查找规则应该与名字所引用的是什么无关。

如果所有这些规则都成立，这个语言将能够以合理的高速度进行语法分析，而且用户将不需要操心这些规则，因为编译程序将能够捕捉到那些具有歧义性或接近歧义性的情况。当前规则已经很接近这个目标了。

1. ARM名字查找规则

在ARM里，我在对付这个问题上取得了一定的成功。来自外层作用域的名字可以直接使用，我还试图通过两个规则去尽可能减少产生顺序依赖性的可能性：

- 1) 类型重新定义规则：如果一个类型名字已在某个类中使用过，就不能在那里重新定义了。
- 2) 重写规则：对于以在线方式定义的成员函数进行分析时，就像它们是在类声明之后定义那样去做。

重新定义规则认定类Y的定义是一个错误：

```
typedef char* T;

class Y {
    T f() { T a = 0; return a; }
    typedef int T; // error T redefined after use
};
```

重写规则说，类X应该被看成是：

```
int x;

class X {
    int f();
    int x;
};

inline int X::f() { return x; } // returns X::x
```

可惜，并不是所有的例子都这么简单。考虑：

```
const int i = 99;

class Z {
    int a[i];
    int f() { return i; }
    enum { i = 7 };
};
```

依照ARM的规则，且（很清楚？）与它们的意图相反，这个例子却是合法的，其中的两个 *i* 引用的是不同的声明，将产生不同的值。重写规则保证在 *Z::f()* 里使用的 *i* 是具有值7的 *Z::i*。但却没有有关 *i* 作为下标使用时的重写规则，所以第一个 *i* 将引用全局的 *i*，具有值99。甚至当 *i* 被用于定义类型时，它本身也不是类型的名字，因此也就不会受到类型重定义规则的管辖。而ANSI/ISO规则将保证这个例子是非法的，因为 *i* 在被使用之后又重新定义了。

同样：

```
class T {
    A f();
    void g() { A a; /* ... */ }
    typedef int A;
};
```

假定在 *T* 之外没有类型 *A* 的定义，*T::f()* 的声明合法吗？*T::g()* 的声明合法吗？ARM规则说 *T::f()* 的声明不合法，因为在那一点 *A* 还没有定义，ANSI/ISO规则也一样。在另一方面，ARM规则可以认为 *g()* 的定义是合法的，如果你把重写规则解释为在语法分析之前的重写；也可以认为它是非法的，如果你解释说语法分析是先做的，而重写在后。这里的问题是：在进行语法分析时 *A* 到底算不是一个类型。我认为ARM规则支持的是第一种观点（也就是说，*g()* 的声明是合法的），但是我无法说明这就是毋庸置疑的和明确的。ANSI/ISO规则同意我对ARM规则的解释。

2. 为什么允许前向引用

原则上说，通过强调严格的一遍分析方式可以避免所有这些问题。你可以用一个名字当且仅当它已经在“上面/前面”被声明了，而任何出现在“下面/后面”的东西都不能对一个声明产生任何影响。这毕竟是C语言的原则，也是C++在其他部分所采用的原则。例如：

```
int x;

void f()
{
    int y = x; // global x
    int x = 7;
    int z = x; // local x
}
```

但在我第一次定义类和在线函数时，Doug McIlroy就确信无疑地争辩说，将这个规则施用到类声明上将造成严重的混乱。例如：

```
int x;

class X {
```

```

void f() { int y = x; } // ::x or X::x ?
void g();
int x;
void h() { int y = x; } // X::x
};

void X::g() { int y = x; } // X::x

```

当x非常大的时候，存在x的不同定义这件事常常会没有被注意到。更糟的是，除非对成员x的使用是一致的，否则重新安排成员的顺序就会导致意义的改变。把一个函数的体移出类声明，作为一个独立的成员函数声明也可能默默地改变了它的意义。前面的重写和重新定义规则提供了一种保护，以防止这种微妙错误，又为类声明的重新组织提供了一些自由度。

这些讨论中也用到了一些非类的例子，但是只有对于类，有关这类保护的编译开销是能够负担的——而且也只有对类这样做时才可能避免与C的兼容性问题。进一步说，类声明也正是最经常发生重新安排、最容易出现不希望有的副作用的地方。

3. ANSI/ISO名字查找规则

在这些年里，我们发现了许多不能由显式的ARM规则概括的例子，它们以很模糊的或者具有潜在危险的方式对顺序有依赖关系，或者是按照有关规则无法做出确定的解释。有的例子真正是病态的，最引人注目的是Scott Turner所发现的一个：

```

typedef int P();
typedef int Q();
class X {
    static P(Q); // define Q to be a P.
    // equivalent to "static int Q()"
    // the parentheses around Q are redundant

    // Q is no longer a type in this scope

    static Q(P); // define Q to be a function
    // taking an argument of type P
    // and returning an int.
    // equivalent to "static int Q(int())"
};

```

在同一个作用域里可以声明具有同样名字的两个函数，只要它们在参数类型上有足够的差异。交换成员函数定义的顺序，我们就定义了两个都称为P的函数；而如果去掉上下文中有关P或者Q的typedef，我们就会得到另一种意思。

这个例子足以使每个人确信，标准化工作对于你的头脑健康是很危险的。我们最后采纳的规则将使这个例子成为无定义的。

与其他许多例子一样，这个例子也是基于由C语言继承来的“隐含的int规则”。我在十多年以前就希望能摆脱这个规则（2.8.1节）。不幸的是，并不是所有病态的例子都源于隐含的int规则。例如：

```

int b;
class Z {

```

```

static int a[sizeof(b)];
static int b[sizeof(a)];
};

```

这个例子是个错误，因为b在被使用之后又改变了意义。幸运的是这类错误很容易被编译程序捕捉到，不像P(Q)那个例子。

在1993年的Portland会议上，委员会采纳了下面的规则：

- 1) 在一个类里一个被声明的名字的作用域不仅包含跟随这个名字声明之后的正文，还包括所有的函数体、默认参数，以及在这个类里的构造函数初始式（包含嵌套的类的这些部分）。但不包括这个名字本身的声明式。
- 2) 在类S里使用过的一个名字，在它的上下文中或在S的完整作用域中重新求值的时候引用的都必须是同一个声明。S的完整作用域由类S本身、S的基类以及所有包含着S的类组成。这通常称为“重新考虑规则”。
- 3) 如果在一个类里重新排列了成员的声明，产生的是另一个符合1)和2)的合法程序，那么该程序的意义是无定义。这通常被称为“重新排序规则”。

请注意，极少有程序会受到这些规则改变的影响。新规则基本上是对原来意图的一种更清晰的陈述。初看起来，这些规则似乎要求在C++实现中使用某种多遍的算法。但实际上它们可以实现为一个一遍算法，以及随后对在这一遍中收集到的信息做一遍或几遍处理。这不会成为性能的瓶颈。

6.3.2 临时量的生存期

C++里的许多运算都需要使用临时量。例如：

```

void f(X a1, X a2)
{
    extern void g(const X&);
    X z;
    // ...
    z = a1+a2;
    g(a1+a2);
    // ...
}

```

一般来说，在给z赋值之前需要有一个（类型X的）对象来保持a1+a2的结果。与此类似，也需要有一个对象保持a1+a2的结果以传递给g()。假定X是一个有建构函数的类，那么在随后的哪个地方对这种临时量调用析构函数？原来我对这个问题的回答是：“在这个块的结尾处，与其他局部变量一样。”已经证明这个回答存在着两个问题：

- 1) 有时这种方式给临时量的生存期不够长。例如，g()可能将指向其参数（由a1+a2产生的临时量）的指针压入一个堆栈，而后可能在f()返回之后的某个地方弹出这个指针并使用该临时量。
- 2) 有时这种方式造成临时量的生存期过长。例如X可能是一个1000乘1000的矩阵，在达到块结束之前可能有数十个临时量被建立起来，这就可能耗尽了本来很大的实际存储器，从而导致虚拟存储机制痉挛式地交换页面。

在我的经验里，前一种情况极少成为真正的问题，对它的答案只能是使用自动废料收集

系统（10.7节）。而后一个问题则是更常见而且严重的。在实践中它将迫使人们把每个可能产生临时量的语句包在一个只包含它自己的块里：

```
void f(X a1, X a2)
{
    extern void g(const X&);

    X z;
    // ...
    {z = a1+a2;}
    {g(a1+a2);}
    // ...
}
```

采用以块结束作为析构点的方式——就像Cfront所实现的那样——用户至少也得做一些显式工作去绕过这个问题。当然，一些用户早就大声疾呼，要求一个更好的解决方案。因此我在ARM里放松了规则，允许在临时量值的第一次使用和块结束之间的任何一点做析构。这是一种意图友善但却是导向错误的行动，它带来了混乱，也没为任何人提供帮助，因为不同的实现者可能为临时量选择不同的生存期。无法写出保证能移植的代码，除非是假定临时量立即被析构——而这很快就被证明是不能接受的，因为它破坏了某些使用频繁、大家都很喜欢的C++习惯用法。例如：

```
class String {
    // ...
public:
    friend String operator+(const String&, const String&)
    // ...
    operator const char*(); // C-style string
};

void f(String s1, String s2)
{
    printf("%s", (const char*)(s1+s2));
    // ...
}
```

在这里的想法就是调用String的转换运算符去产生一个C-风格的字符串，再交给printf打印出来。在典型的（朴实有效的）实现里，转换运算符就是简单地返回一个指向String对象中的一部分的指针。

如果转换运算符采用这种简单实现方式，这个例子在“立即析构临时量”的规则下就无法工作了：为s1+s2建立起一个临时量，下一步得到C风格字符串的方式就是取得一个指向这个临时量内部的指针，在临时量被析构之后，指向这个已经销毁的临时量内部的指针又被传给了printf。对保有s1+s2的String临时量的析构当然应该已经释放了保有这个C风格字符串的存储区。

这种代码太常见了，甚至那些一般都遵循立即析构策略的C++实现，例如GNU的C++，对于这种情况也要推迟析构。这种考虑就导向了另一种想法：在建立起临时量的语句的最后去销毁它们。这样做将能使上面例子合法化，而且保证了跨实现的可移植性。但是另外的“几乎等价”的例子仍将崩溃，例如：

```
void g(String s1, String s2)
{
```

```

const char* p = s1+s2;
printf("%s", p);
// ...
}

```

按照“在语句结束处销毁临时量”的策略，由p指向的C风格字符串（它实际驻在代表s1+s2的临时量里）将在初始化p的语句结束处被释放掉。

关于临时量生存期的讨论在标准化委员会里持续了大约两年时间，直到Dag Brück成功地结束了它。在此之前，委员会花了大量时间讨论各种都是足够好的解决方案的相对优点，每个人都认为并不存在完美的解。我的意见——有时是大声疾呼——是用户已经受到了没有解决方案的伤害，现在已经到了必须确定一个的时候了。我认为最好的办法就是选出一个来。

Dag在1993年7月对这个问题做了总结，基本上是基于Andrew Koenig、Scott Turner和Tom Pennello的工作。在总结中标出了临时量析构的7个主要的可能点：

- 1) 在第一个使用之后。
- 2) 在语句结束处。
- 3) 在下一个分支点。
- 4) 在块的结束处（原来的C++规则，如Cfront）。
- 5) 在函数的结束处。
- 6) 在最后的使用之后（隐含着废料收集）。
- 7) 在第一次使用和块结束之间，不给以定义（ARM规则）。

可以构造出有利于每种选择的合法论据，我把这留给读者，作为一个练习，这是可以完成的。当然，对每一个也可以给出认真的合法的反面证据。因此，实际问题就是要做出一种在各种利益和问题之间取得较好平衡的选择。

此外，我们还考虑了在块中的最后一次使用之后销毁临时量，但是这就需要做控制流分析，我们都认为不能要求每个编译程序都能很好地完成控制流分析，以保证在每个实现里“在块中的最后一次使用之后”都是计算过程中一个定义明确的点。请注意，局部流分析是不够的，它无法合理地提供“过早析构”的警告信息。返回一个到对象内部的指针的转换函数也常常被定义在某个编译单位里，可能与使用这种函数的编译单位不同。想禁止这种函数是不行的，这样做将打破许多现存的代码，也不可能强制地推行。

从大约1991年开始，委员会将注意力集中到“语句结束位置”，很自然地，这种选择被通俗地称作EOS (End Of Statement)。问题是如何精确定义EOS的意义。例如：

```

void h(String s1, String s2)
{
    const char* p;

    if (p = s1+s2) {
        // ...
    }
}

```

变量p的值可以合法地在语句块里使用吗？也就是说，对于保存s1+s2的对象的析构是应该在条件的最后进行呢？还是应该在整个if语句的最后？回答是：保持s1+s2的对象将在条件的最后被析构。要保证下面的东西是很不合理的：

```
if (p = s1+s2) printf("%s",p);
```

如果已经说

```
p = s1+s2;
printf("%s",p);
```

依赖于实现的话。

对于一个表达式里的不同分支应该如何处理？例如，下面这个应该保证工作吗？

```
if ((p=s1+s2) && p[0]) {
    // ...
}
```

回答是应该。解释这个回答要比解释有关`&&`、`||`和`? :`的特殊规则容易得多。对这点也有一些负面因素，因为这种规则的一般性实现很不容易做，除非是引进标志去保证，只有当临时对象出现在实际执行的分支上时它们才会被销毁。但是委员会里的编译专家站起来响应这个挑战，证明了由此引起的开销是极小的，基本上不值得一提。

这样EOS就变成意味着“完整表达式的结束”，而所谓完整的表达式就是说它不再是其他表达式的子表达式了。

请注意，采用在完整表达式结束时销毁临时量的解决方案将会打破一些Cfront代码，但是它却不会打破任何能够保证在ARM规则下工作的代码。这种方案满足了对一种定义良好且容易解释的析构的需要。它也满足了另一种需要：不要让临时量闲置在那里太长时间。如果需要生存得更长些的对象，就必须予以命名，或者是换一种方式，采用某种不需要更长生存期的对象的技术，例如：

```
void f(String s1, String s2)
{
    printf("%s",s1+s2); // ok

    const char* p = s1+s2;
    printf("%s",p); // won't work, temporary destroyed

    String s3 = s1+s2;
    printf("%s", (const char*)s3); // ok

    cout << s3; // ok

    cout << s1+s2; // ok
}
```

6.4 扩充

那时最关键的一个问题是——现在依然是——如何处理源源不断的有关语言修改和扩充的建议。集中处理这个问题的是扩充工作组，我是主席。接受一个建议比拒绝它要容易得多。这样做你赢得了朋友，人们也为语言里有了这么多“好特征”而鼓掌欢呼。但如果把一个语言做成了特征的清单，没有内在的一致性，它将注定要死亡。所以我们没有办法，甚至无法接受大部分一定会给C++社团的某些部分带来真正帮助的特征。

在Lund（瑞典）会议上，下面这段警示性的故事流传开了：

“我们常常提醒自己不要忘记良舰Vasa，那时它是瑞典海军的骄傲，计划建造成有史以来最大最美丽的战舰。不幸的是，为了装备足够多的雕像和大炮，在它的建造过程中经历了重大的重新设计和扩充。结果是它只驶过了斯德哥尔摩湾的一半，一阵风吹过来，它就翻了个底朝天。它沉没了并且杀害了大约50个人。它后来被打捞起来，你可以在斯德哥尔摩的一个博物馆里看到它。它看起来异常美丽——那时就比它那从未存在过的第一次设计要美丽得多；与假设它遭受过17世纪战舰的通常命运的情况相比，它今天就美丽得太多了。可是，对于它的设计者、建造者和预期的使用者而言，这些都根本不能作为一种安慰 [Stroustrup, 1992b]。”

那么究竟为什么要考虑扩充呢？X3J16终归是个标准化组，不是获准去设计“C++++”的语言设计组。更糟的是，这是一个有多于250个成员的组，其成员还在随着时间变化，这种组织根本不是设计语言的合适场所。

首先，这个组织已被指示去处置模板和异常处理问题。甚至在委员会有时间开始这个工作之前，有关扩充甚至不兼容修改的建议就已经寄到了委员会成员那里。用户社团，甚至那些并没有提出建议的大部分用户，也都很清楚地希望委员会考虑这些建议。如果委员会认真地考虑这些建议（正如它实际所做的那样），它也就是提供了一个集中讨论C++特征的场所。如果它不这么做，有关的活动也会在其他地方进行，结果将是出现各种互不兼容的扩充。

此外，虽然人们嘴上都说需要最小化和稳定性，大部分人实际上是很喜欢新特征的。语言设计从本质上就是有趣的，有关新特征的争论使人兴奋，它们为新论文和新的发布提供了口实。有些特征甚至也可能有助于程序员，但对许多人来说这不过是第二位的推动力。如果被忽略，这些因素也可能阻碍进步。我更希望它们能有一些建设性的产出。

这样，实际上委员会可以在讨论这些扩充，或者讨论扩充被使用后产生的方言，或者忽视现实之间做一个选择。在这些年里，这些选择中的任何一个都曾被不同的委员会选用过。大部分委员会——包括Ada、C、Cobol、Fortran、Modula-2和Pascal-2委员会——都选择去考虑扩充。

按照我个人的观点，各种类型的扩充活动是不可避免的。最好是让它公开，将其引导到某种形式规则之下；以一种半文明的方式在一个公开的论坛中进行。另一种选择就是竭力搜寻，通过在市场上吸引用户的方式去取得可接受的想法。这种方式不会有助于平静的思考、公开的讨论和服务于所有用户的努力。

我认为，与不处理扩充而导致的混乱局面相比，由处理扩充引起的明显的内在危险还更可取些。在委员会里慢慢积累起来的多数都已经同意，我们已经接近了这样一点，至今在我们的引导下进行的扩充工作必须逐渐停息下来，因为标准文档将开始出现了，所有活动都必须直接面向有关这些文档的各种反应。

只有时间能告诉我们，这样遗留下来尚未释放的能量会奔向何方。有些将进入其他语言，有些将变成试验性工作，有些将转变为库的构造（这是传统的C++替代语言改变的另一种方式）。看起来很有趣，标准化组也像其他组织一样，会发现要解开自己的束缚是非常困难的。通常一个标准化组重构自己的方式是把自己变成一个完成修订的论坛，或者变成一个为建立下一层标准的官僚机制，也就是说，成为一个新的语言或者方言的设计委员会。Algol、Fortran和Pascal委员会，甚至ANSI C委员会都是这种现象的实例。常见情况是，从一个标准化了的语言出发，重新定向去设计一个所谓的后继语言的工作，总需要伴随着人员以及想法

方面的一个大转变。

在此期间我试图扮演一个卫士的角色，防止由委员会进行设计所产生的危险：把太多时间花费在各个扩充提议上。所采用的策略并不是简单明了的，但它确实能提供一定程度的保护，防止接受互相冲突的特征，防止丢掉语言的内在一致性。

由委员会进行设计的主要危险就是丢掉了有关被设计的语言是什么，应该如何发展的具有内在一致性的观点，而转变成为针对个别语言特征或者解决方案的政治交易。

一个委员会很容易陷入陷阱去赞同一个特征，只不过是因为某些人强调这个东西是非常关键的。为一个特征争辩，说明它的优点——通过某些令人感兴趣的例子——做起来比较容易；而要说明它的优点被高估了——在一致性、简单性、稳定性、翻译的困难程度等方面都没有考虑清楚——则更困难得多。此外，语言委员会的工作方式好像也不利于在试验和实际经验的基础上争论问题。我不太清楚为什么会这样，可能是委员会的形式和通过投票解决激烈争论问题的方式更容易被筋疲力尽的成员们所接受吧。还有一点也很清楚，逻辑的理由（有时甚至是非逻辑的争辩）也比基于别人经验和试验的报告更有说服力。

这样，“标准化”就有可能变成一种趋向于不稳定的力量。这种不稳定性的结果也有可能转变为一个更好的东西，但却始终存在着导致某种随机转变、甚至变出一种更坏的东西的危险性。为了避免这类情况，标准化必须成为语言发展过程中一个正确的台阶：在已经勾勒出它的发展道路的轮廓之后，在由强有力的商业利益所支持的方言出现之前。我希望这就是C++的情况，而这个委员会将要继续在改革中显示出它必要的约束力。

此外，值得一提的是，即使没有扩充人们也照样能过得去。语言特征中相当大的一部分通常都被人们忘记了，没有巧妙语言的支持要构造出好的软件也是很可行的。没有任何一个单独的语言特征对于好的软件设计而言是必需的，甚至包括那些我们绝不想丢掉的东西。好软件可以，而且常常是用C或者C++的一个小子集写出来的。语言特征的价值在于为表述思想所提供的方便性，把一个程序做正确所需要的时间，结果代码的清晰性，以及结果代码的可维护性。用被指责为“坏”的语言写出的好代码比用断言是“美妙”的语言写出的好代码更多——多得多。

6.4.1 评价准则

为了帮助人们理解在计划对C++做一个扩充或者改变时涉及到哪些事情，扩充工作组提出了一些问题，通常对每个建议的新特征都需要问这些问题 [Stroustrup, 1992b]：

“这个列表给出了一些准则，用来评价有关C++的特征。

(1) 它精确吗？（我们是否能了解你所建议的是什么？）写出有关这个改变的一个清晰、精确的陈述，说明它对当前的语言参考标准的影响。

- (a) 语法上需要做什么修改？
- (b) 对语言语义的描述需要做什么修改？
- (c) 它是否能够与语言的其他部分相配合？

(2) 这个扩充的理由是什么？（为什么你希望有它？为什么我们也会希望有它？）

- (a) 为什么需要这个扩充？
- (b) 谁欢迎这个修改？
- (c) 这是一个用途广泛的修改吗？

- (d) 它是不是对某一组C++语言用户的影响比对其他人的影响更大?
- (e) 它在所有合理的硬件和系统中都可以实现吗?
- (f) 它在所有合理的硬件和系统中都有用吗?
- (g) 它支持的是哪一类的编程和设计风格?
- (h) 它阻止的是哪一类的编程和设计风格?
- (i) 哪些其他语言(如果有的话)提供了这些特征?
- (j) 它能有助于库的设计、实现和使用吗?

(3) 它已经被实现了吗? (如果实现了, 那么它是否完全是按照你所建议的形式实现的? 如果没有实现, 那么为什么你能假定“类似”实现或者其他语言的经验足以移到到所建议的特征上?)

- (a) 它对于一个C++实现将有什么影响?
 - (x) 对于编译程序的组织?
 - (y) 对于运行时的支持?
- (b) 这样的实现完成了吗?
- (c) 除了实现者自己之外, 还有其他人使用过这个实现吗?

(4) 这个特征对代码有什么影响?

- (a) 如果没有这个修改, 代码将是什么样的?
- (b) 如果不做这个修改会有什么影响?
- (c) 使用新特征是否将导致对新工具的要求?

(5) 这个修改对于效率和与C语言和现在的C++的兼容性有什么影响?

- (a) 这个修改对运行效率有什么影响?
 - (x) 对于使用这个特征的代码?
 - (y) 对于不使用这个特征的代码?
- (b) 这个修改对编译和连接时间有什么影响?
- (c) 这个修改是否影响现存的程序?
 - (x) 没使用这个特征的C++程序必须重新编译吗?
 - (y) 这个修改是否影响与其他语言, 如Fortran或者C的连接?
- (d) 这个修改对于C++程序的静态或动态检查的可能程度有影响吗?

(6) 这个修改的文档及教学简单吗?

- (a) 对于新手?
- (b) 对于专家?

(7) 可能存在哪些理由支持不做这种扩充? 肯定存在反对的意见, 而我们工作的一部分就是发现和评价它们, 因此如果你给出一个讨论, 就能节省时间。

- (a) 它是否影响那些不使用新特征的代码?
- (b) 它是否很难学?
- (c) 它是否会引出进一步的扩充需要?
- (d) 它是否会导致很大的编译程序?
- (e) 它是否要求扩充的运行支持?

(8) 是否存在

- (a) 能够服务于这个需要的其他替代性特征?
- (b) 对于所建议的语法的其他替代形式?
- (c) 针对所建议模式的有吸引力的更一般的替代形式?

自然,这个列表并不完全。请扩充它,使之包括某些与你的特定建议有关的论点,并不必理会那些无关的问题。”

这些问题当然是对实际语言设计者们总去问的那类问题的一个汇编。

6.4.2 状况

那么委员会又做了些什么呢?在标准出来之前我们不可能确切地知道,因为无法知道新建议将来会怎么样。这个总结是基于1994年11月在Valley Forge时的情况。为C++提供的扩充建议范围非常广泛,例如:

- 扩充的(国际)字符集(6.5.3节)
- 各种模板扩充(15.3.1节, 15.4节, 15.8.2节)
- 废料收集(10.7节)
- NCEG^①的建议(例如6.5.2节)
- 带区分符的联合
- 用户定义运算符(11.6.2节)
- 可发展的/间接的类
- 带有<<、++等运算符的枚举
- 基于返回类型的重载
- 复合运算符(11.6.3节)
- 表示空指针的关键字(NULL、nil等)(11.2.3节)
- 前/后条件
- 对Cpp宏的改进
- 引用的重新约束
- 继续
- Curry化^②

存在着对它们进行限制的希望,被接受的特征将要很好地集成进语言里。迄今为止只有不多的一些特征被接受了:

- 异常处理(“委托事务”)(第16章)
- 模板(“委托事务”)(第15章)
- C++的欧洲字符集合(6.5.3节)
- 对于重载函数返回类型的放松了的规则(13.7节)
- 运行时的类型识别(14.2节)
- 条件里的声明(3.11.5节)
- 基于枚举的重载(11.7.1节)

① Numerical C Extension Group, C语言数值扩充组织。——译者注

② 有关函数参数的一种变形规则。——译者注

- 针对数组的用户定义分配和释放运算符 (10.3节)
- 嵌套类的前向声明 (13.5节)
- 名字空间 (第17章)
- 变化量 (`mutable`) (13.3.3节)
- 布尔类型 (11.7.2节)
- 类型转换的一种新语法形式 (14.3节)
- 模板函数调用的显式模板参数 (15.6.2节)
- 成员模板 (15.9.3节)
- 类模板作为模板的参数 (15.3.1节)
- 整型类的`const static`成员可以在类声明中用一个常量表达式进行初始化
- 显式地使用建构函数 (3.6.1节)
- 对异常规范的静态检查 (16.9节)

在这些扩充中，异常和模板是在ARM里提出建议和描述的委托事务，从定义和实现的难度看它们也比任何其他提议都困难几个数量级。

相应地，委员会也拒绝了大量的建议。例如：

- 若干有关直接支持并发性的建议
- 对继承而来的名字重新命名 (12.8节)
- 关键词参数 (6.5.1节)
- 关于对数据隐蔽规则做小修改的一些建议
- 受限指针 (“noalias的儿子”) (6.5.2节)
- 指数运算符 (11.5.2节)
- 自动生成的复合运算符
- 用户定义的`operator .()` (11.5.2节)
- 嵌套的函数
- 二进制字面量
- 在类声明里对成员的普遍初始化规则

请注意，拒绝并不意味着这个建议就被认为是不好的甚至无用的。实际上，递交给委员会的许多建议在技术上都是有效的，而且至少会对C++社团的一部分人有所帮助。原因是大部分想法本身都没有经过初步的认真检查，没有努力做成建议。

6.4.3 好的扩充问题

即使是好的扩充也会带来问题。假定在某个时刻我们有了一个人都喜欢的扩充，因此讨论它的合法性绝不会是浪费时间。它也会吸引实现者的精力，使他们离开一些人们认为是非常重要的工作。例如，一个实现工作者可能要做出选择，是去实现某种新特征呢，还是实现在代码生成程序里的一个优化。经常是新特征取胜，因为它对于用户而言能够看得更清楚。

一个扩充孤立地看可能是完美无缺的，但在某种更大范围下观察可能就露出了瑕疵。对一个扩充的许多工作都集中在将它集成到语言之中，集中到该特征与其他语言特征的相互关系上。这类工作的困难程度和为把它做好所需要的时间总是被人们低估。

任何新特征都会使现存的实现变得过时了，因为它们没有处理新的特征。这样，用户也

将不得不做些升级工作：在没有新特征的情况下生活一段时间，或者同时管理系统的两个版本（一个用于最新的实现，一个用于老的实现）。后面这种情况常常是库或者工具的制造者们必须做的事情。

教学材料也需要更新，以便能反映新特征的情况，与此同时可能还需要反映原来使用的语言，这是为了那些还没有更新的用户的利益。

一个完美的扩充也有负面作用。如果所建议的扩充是有争议的，它将从委员会成员以及整个委员会那里进一步榨掉许多精力。如果扩充有不兼容的方面，在从老的实现更新到新实现时就会遇到一些问题——有时甚至在你并没有使用新特征的时候。一个经典的例子是引进一个新关键字。例如，下面是一个很普通的查找函数：

```
void using(Table* namespace) { /* ... */ }
```

在引进了名字空间之后，它就不再合法了，因为using和namespace都已成为新的关键字。按照我的经验，虽然引进新关键字制造出一些新技术问题，但这些问题还是很容易修正的。从另一方面说，建议一个新关键字不会引起感到被凌辱而产生的嚎叫。新关键字所带来的实际问题可以尽可能地减小，方法是适当地选择名字，使之不太可能与现存的标识符冲突。正是由于这个原因，应该使用using而不是use，选择了namespace而不用scope。作为试验，我们把using和namespace引进一个局部实现，经过两个月时间居然没有人注意到它们的存在。

除了使一个新特征被接受并被使用可能涉及到的各种实际问题之外，仅仅是有关扩充的讨论也会产生负面影响，它往往给人一种某些人的思想不太稳定的想法。许多用户和自称的用户并不理解这些修改已经经过了仔细的甄别，以尽可能减少对现存代码的影响。新特征的理想主义的辩护者们则常常认为稳定性和与C语言及现有C++兼容的限制很难接受，在触发对不稳定的恐惧方面做得太多了。还有，那些“改革”的狂热鼓吹者们也倾向于夸大语言的弱点，以便使他们的扩充看起来更具有吸引力。

6.4.4 一致性

我认为，扩充建议所面临的主要挑战在于维持C++的内在一致性，以及如何把这种一致性的观点传播给用户社团。被接受进C++的特征必须能够组合在一起工作，必须能互相支持，必须能弥补起一些在没有这种扩充之前C++里存在着的严重的现实问题，必须能在语法上和语义上融入语言之中，还必须能支持一种可以把握的程序设计风格。一个程序设计语言绝不能只是一个美好特征的汇集，在评价和开发扩充时，最主要的精力应该用在精化它们方面，使之能变成语言中一个完美的组成部分。对一个我认真考虑了的扩充，我把95%的个人精力花在为原始的思想/建议寻找一种合适的形式，使它能平滑地集成到C++里。典型情况是，这其中的大部分又涉及到如何为实现者和用户找到一条清晰的转变之路。即使是最好的新特征，如果用户不抛弃他们的大部分老代码和老工具就不能接纳它的话，这个特征也是必须拒绝的。参看第4章有关接受准则的范围更广泛的讨论。

6.5 扩充建议实例

在本书里，一般地说我都在相关特征的上下文里讨论所建议的语言特征。但也有几个东

西放在哪里都不太合适，因此我把它们作为这里的例子。不会令人奇怪，这些无法自然地与任何地方匹配的东西当然更可能会被拒绝。一个特征，无论独立地看起来多么合理，都应该抱着极大的疑问，除非它能被看作是在语言的某个既定发展方向上的更一般性努力的一个组成部分。

6.5.1 关键词参数

Roland Hartinger提出了关于关键词参数的建议，这是在函数调用中使用的一种参数描述机制。这个建议在技术上几乎是完美无缺的。因此，这样的建议被拒之门外而没有接受就特别有意思。这个建议被退回是因为扩展小组达成了一个共识：这个建议近乎多余，可能导致与现存C++代码的兼容性问题，也可能鼓励一种不应该的程序设计风格。下面的讨论反映了在扩充工作组里的讨论。当然，因为篇幅所限，成百的有关意见不可能在这里一一描述。

考虑一个丑陋的（很不幸，它并不是不现实的）例子，它来自Bruce Eckel写的一个分析报告：

```
class window {
    // ...
public:
    window(
        wintype=standard,
        int ul_corner_x=0,
        int ul_corner_y=0,
        int xsize=100,
        int ysize=100,
        color Color=black,
        border Border=single,
        color Border_color=blue,
        WSTATE window_state=open);
    // ...
};
```

如果你想定义一个默认的window，那么一切都很好。但如果你想定义一个“几乎是”默认方式的window，其描述就可能冗长乏味，而且很容易出错。这个建议就是简单地引进一个新运算符:=，允许把它用在调用里，为某个指名的参数提供值。例如：

```
new window(Color:=green,ysize:=150);
```

将等价于：

```
new window(standard,0,0,100,150,green);
```

如果考虑默认参数，这个描述将等价于：

```
new window(standard,0,0,100,150,green,single,blue,open);
```

这个东西看起来像是一种很有用的语法糖衣，能够使程序更容易读也更坚固。这个建议已经实现了，可以保证消除了所有概念方面和集成方面的问题，也没有发现明显的或困难的问题。进一步说，所建议的这个机制也是基于其他语言的经验，例如Ada等。

另一方面，我们在没有关键词参数的情况下无疑也能够生活。这个建议并没有提供任何新的基本概念，不支持任何新的程序设计范例，也不能堵住类型系统里的任何漏洞。这样就留下了几个需要回答的问题，要更多地看C++用户社团现时的口味和印象：

- 1) 关键词参数能导致更好的代码吗?
- 2) 关键词参数会导致混乱, 或给教学带来问题吗?
- 3) 关键词参数会导致兼容性问题吗?
- 4) 关键词参数应该成为我们能够接受的不多的几个扩充之一吗?

对于这个建议, 发现的第一个严重问题是关键词参数引进了调用界面和实现之间的一种新的约束形式:

- 1) 在函数声明里的每个参数都必须采用与函数定义里一样的名字。
- 2) 一旦使用了某个关键词参数, 在函数定义里的参数名字就无法更改, 否则就会打破用户代码。

由于重新编译的巨大代价, 许多用户对加强定义界面和实现之间的约束关系总是心怀疑虑。更糟的是, 这还会成为一个重大的兼容性问题。某些机构建议了一种风格, 在头文件里使用“长而富含信息的”参数名字, 而在函数定义中使用“短而方便的”的名字。例如:

```
void reverse(int* elements, int length_of_element_array);

// ...

void reverse(int* v, int n)
{
    // ...
}
```

当然, 有些人会认为这种风格让人恶心, 但是另外一些人(包括我)觉得它也是相当合理的。很明显的是现存大量的这种代码。进一步说, 关键词参数还意味着一个广泛散布的头文件将不能被修改, 否则就有打破许多代码的危险。一种公共服务(例如Posix或者X)的不同提供者也必须在参数名字上达成一致意见。这很容易变成一个官僚主义的噩梦。

如果换一种方式, 语言可以不要求声明中对同一个参数使用同样的名字。对我来说这也还可以接受, 但是人们不大会喜欢这种变形。

如果必须在编译单元之间检查参数名字的匹配问题, 这种改变就将对连接所用的时间产生重大的影响。如果不检查, 这种机制就不是类型安全的, 很可能成为微妙错误的根源。

对于这种潜在的连接代价以及非常实际的连接问题, 人们避免它们的一种简单方式就是在头文件里完全不写参数名字。对此Bill Gibbons说: “这对于C++可读性的深远影响将是负面的。”

我对关键词参数的主要忧虑, 在于它实际上可能延缓在C++里人们从传统的程序设计技术向数据抽象和面向对象的程序设计的转变。我认为, 在写得最好最容易维护的代码中, 出现超长参数表的情况是很罕见的。实际上人们有一个共同的认识, 转变到面向对象的风格也将导致参数表的显著缩短, 因为一些往常是参数或者全局的值现在却变成局部状态了。基于这些经验, 我预期平均的参数表长度将下降到2以下, 而超过两个参数的情况将变成罕见的。这也就意味着关键词参数仅仅是在那些我们认为写得非常糟糕的代码里才有用。引进一个新特征只是为了去支持一种我们都希望消灭的程序设计风格, 这难道是合情合理的吗? 最后的共识是不, 这个结论基于上面这些讨论, 兼容性问题, 以及其他一些不太重要的细节。

关键词参数的变形

在没有关键词参数的情况下, 我们怎样才能缩短在window例子里参数表的长度, 使之变

为某种方便的形式呢？首先，通过默认参数，复杂性已经明显地降低了。增加其他类型也是表示常见变形的一种惯用技术：

```
class colored_window : public window {
public:
    colored_window(color c=black)
        :window(standard, 0, 0, 100, 100, c) { }
};

class bordered_window : public window {
public:
    bordered_window(border b=single, color bc=blue)
        :window(standard, 0, 0, 100, 100, black, b, bc) { }
};
```

这种技术有很多优点，它把使用方式引导到一些规范的形式上，因此能使代码和行为更加规范。另一种技术是提供一个明显操作去修改对默认参数的设置：

```
class w_args {
    wintype wt;
    int ulcx, ulcy, xz, yz;
    color wc, bc;
    border b;
    WSTATE ws;
public:
    w_args() // set defaults
        : wt(standard), ulcx(0), ulcy(0), xz(100), yz(100),
        wc(black), b(single), bc(blue), ws(open) { }

    // override defaults:

    w_args& ysize(int s) { yz=s; return *this; }
    w_args& Color(color c) { wc=c; return *this; }
    w_args& Border(border bb) { b = bb; return *this; }
    w_args& Border_color(color c) { bc=c; return *this; }
    // ...
};

class window {
    // ...
    window(w_args wa); // set options from wa
    // ...
};
```

从这里我们能得到一种很方便的记法，它大致上等价于关键词参数能够提供的东西：

```
window w; // default window
window w( w_args().Color(green).ysize(150) );
```

这种技术具有重要优点，它使得在一个程序里传递由对象表示的参数变得更容易了。

当然这些技术也可以组合起来使用。这些技术的作用将大大缩短参数表，也减少了使用关键词参数的必要性。

进一步减少参数的另一种方式是采用一个Point类型，而不是直接以坐标形式来描述

接口。

6.5.2 受限指针

Fortran编译程序允许做出这样的假定：如果传递给函数两个数组参数，那么这两个数组相互是没有重叠的。对于C++函数就不能做这个假定。由这个假定导致Fortran子程序得到15%到30倍的加速，具体情况依赖于编译程序的质量和计算机体系结构。在这里令人吃惊的时间缩短情况来自具有特殊硬件机器上的向量化运算，例如在Cray机器上。

C语言是强调效率的，这种情况被认为是一种侮辱，对此ANSI C委员会提出过一种称为noalias的机制来解决这个问题，用它来说明某个C指针可以认为是没有别名的。不幸的是这个建议来晚了，而且又是如此的不成熟。这件事激怒了Dennis Ritchie，使他对C的标准化过程做了惟一的一次干预。他写了一封公开信说“noalias必须靠边站，这一点是不能协商的。”

在那以后，C和C++社团对处理别名问题都非常谨慎。但是这个问题对于Cray机器上的C用户确实非常重要，因此Cray的Mike Holly又抓起了这个难题，向数值C语言扩充工作组(Numerical C Extension Group, NCEG)和C++委员会提出了一种改进的反别名建议。所建议的想法是允许程序员说明一个指针可以认为是没有别名的，采用的方式是将它说明为restrict。例如：

```
void* memcpy(void* restrict s1, const void* s2, size_t n);
```

因为s1已经说明为没有别名的，因此就不再需要说明s2为restrict。关键字restrict对*的作用方式与const、volatile一样。这个建议应该能解决C/Fortran在效率上的差异，只要选择性地采用Fortran的规则就可以了。

作为C++的委员会，当然非常同情任何能够改进效率的建议，因此对这个建议讨论了很长时间，但是最后还是决定拒绝它，几乎没有不同的声音。拒绝的关键理由是：

(1) 这个扩充是不安全的。把一个指针声明为restrict，就是让编译程序假定这个指针没有别名。然而作为用户对此却可能没有充分警觉，而编译程序又不能保证它。由于在C++里指针和引用的使用非常广泛，与Fortran提供的经验相比，在C++里这个因素可能导致更多的错误。

(2) 对于这个扩充的其他选择还没有进行充分的研究。在许多情况下，诸如采用对重叠做初始检查并结合对非重叠数组的特殊代码也可能是一种选择。对另一些情况，可以直接调用特殊的数学库，例如BLAS，以这种方式为提高效率而转到向量代码。另一些为了优化的很有前途的方式也在研究之中。例如，对相对较小具有特定风格的向量和矩阵运算做全局优化看起来是可能做的，研究高性能计算机使用的C++编译程序也很有价值。

(3) 这个扩充也是与机器有关的。高性能数值计算是一个特殊领域，使用特殊的技术，通常在特殊硬件上做。由于这些情况，更合适的方式可能是引进一种为非标准体系结构所使用的特殊扩充或者语用指示词。在这个使用特殊机器体系结构的小用户群之外，对这类优化所需要的这种特殊功能是否也确实非常有用呢？这个扩充必须重新进行评价。

这个决定的一个方面的意义是进一步确认了C++是通过具有普遍性的机制支持抽象的思想，而不是通过专用机制去支持特定应用领域。我也确实想帮助数值计算用户群，但问题是这么做？对于向量和矩阵算法，紧紧跟随Fortran的脚步可能并不是最好的途径。如果能把所

有的各种数值软件都用C++写出来而又不损失效率，这当然是最好的事情。但这就需要发现某些能达到这种情况的东西，而且又不危及C++的类型系统。依靠Fortran、汇编语言或者某种针对特定体系结构的扩充可能是更好的办法。

6.5.3 字符集

C依赖于国际7位字符集ISO 646-1983的美国变形，称为ASCII (ANSI3.4-1968)。这带来两个问题：

(1) ASCII包含标点符号字符和运算符，例如]和{，在许多国家的国内字符集上它们都不能使用。

(2) ASCII不包含在英语之外的其他语言的某些字符，例如Å和æ。

1. 受限字符集

ASCII (ANSI3.4-1968) 特殊字符[、]、{、}和\占据了ISO指定为字母的字符位置。在大部分欧洲国家的ISO-646字符集里，这些字符是由不在英语字母表里的字母占据的。例如，在丹麦国家字母表里用这些值表示元音字母Æ、Å、æ、å、ø和Ø。没有它们就无法写出丹麦语里的任何有实际价值的文本。这个问题使丹麦程序员遇到了一种两难的选择：或者是需要能处理8位字符集（例如ISO-8859/1/2）的计算机系统，同时又不使用自己母语里的三个元音；或者就是不用C++。说法语、德语、意大利语等的人们也都面临着同样的选择。这已经成为在欧洲使用C语言的一个很值得注意的障碍，特别是在那些商务部门（例如银行），在许多国家里这些部门广泛使用7位的国家标准字符集。

例如，考虑下面这个看起来很淳朴的ANSI C和C++程序：

```
int main(int argc, char* argv[])
{
    if (argc<1 || *argv[1]=='\0') return 0;
    printf("Hello, %s\n", argv[1]);
}
```

在标准的丹麦终端或者打印机上，这个程序就是下面的样子：

```
int main(int argc, char* argv[])
{
    if (argc<1 || *argv[1]=='\0') return 0;
    printf("Hello, %s0n", argv[1]);
}
```

如果认为某些人能够流利地阅读和书写这种东西，那才真是奇怪呢。我绝不认为这应该成为对任何人提出的一种技能要求。

ANSI C委员会采纳了一个针对这种问题的部分解决方案，定义了一组三字符序列，这就使那些本国字符也可以表示了：

```
#      [      {      \      ]      }      ^      |      ~
??=    ??(    ??<   ??/    ??)    ??>   ??'    ??!    ??-
```

对于交换程序而言，这种做法很有用，但它却削弱了程序的可读性：

```
int main(int argc, char* arg??(??))
??<
    if (argc<1 ??!??! *argv??(1??)=='??/0') return 0;
```

```
printf("Hello, %s??/n", argv??(1??));
??>
```

自然，真正能为C和C++程序员解决这个问题的办法是购买那种既能支持其本国语言也能支持C和C++所需要字符的设备。不幸的是对某些人而言这肯定是行不通的，引进新设备必然要经过一个漫长的过程。为了帮助程序员使用现有设备，也为了帮助C++本身，C++委员会决定提供另一种更可读的形式。

提供下面这些关键字和二联符序列，在那些包含本国字符的地方作为对应运算符的等价物：

关 键 字	二联符序列
and	&&
and_eq	&=
bitand	&
bitor	
compl	~
not	!
or	
or_eq	=
xor	^
xor_eq	^=
not_eq	!=

我个人更喜欢用%%代表#，用<>表示!=，但%:和not_eq已经是C和C++委员会能同意的最好的东西了。

现在我们就可以把上面的程序写成如下样子：

```
int main(int argc, char* argv<: :>
<%>
    if (argc<1 or *argv<:1:>=='??/0') return 0;
    printf("Hello, %s??/n", argv<:1:>);
??>
```

请注意，为了把“缺少的”字符，例如\，放入字符串和字符常量，三联符序列还是需要的。

引进二联符序列和新关键字引起了许多争论。一大批人——主要是以英语为母语的人们和有着很强C背景的人们说，完全没有必要为了取悦一些“不想去买像样设备”的人而把C++弄得更加复杂而污浊。我赞成这种论点，因为三联符和二联符序列都不漂亮，新关键字也总是不兼容性的一个根源。但在另一方面，我将不得不一直用无法支持我的母语的设备工作，而且我也不得不看着某些人抛弃C语言作为一种可能的程序语言，而转去用另一种“不使用可笑字符的语言”。作为对这种观点的一个支持，IBM的代表报告说，由于在IBM主机所用的EBCDIC字符集里缺少!字符而导致经常地反复地出现了许多抱怨。我还发现了一个很有趣的事，即使是在那些能使用扩充字符集的地方，系统管理员也常常关闭掉它们的使用。

我认为存在一个可能长达10年的转变期，在此期间关键字、二联符和三联符序列是害处最少的解决办法。我的希望是，它能够帮助C++被那些C语言没有渗入的领域所接受，并以此支持那些还没有在C和C++文化中有其代表的程序员们。

2. 扩充字符集

对C++而言，支持一种受限字符集表示了一种向后看的态度。另一个更有趣也更困难的问题是如何支持扩充的字符集，也就是说，如何获得那些具有比ASCII更多字符的字符集的优点。在这里存在两个不同问题：

- 1) 如何支持对扩充字符集的操作？
- 2) 如何允许在C++程序正文里使用扩充集合？

C标准化委员会对前一个问题的答复是定义了一种表示多字节字符的类型wchar_t，此外还提供了一个多字节字符串类型wchar_t[]以及对wchar_t的printf一族的I/O功能。C++在这个方向上继续前进，把wchar_t作为一个真正的类型（而不是像在C里那样，只把它作为通过typedef定义的其他类型的一个同义词），还提供了一个标准的wchar_t的字符串类wstring，并在I/O流中支持这些类型。

这样支持的仅仅是一个“宽字符”类型。如果程序员需要更多的字符，比如说一种日文字符，一种日文字符的字符串，一种希伯莱文字符和一种希伯莱文字符串，那么至少存在两种不同的解决办法。一种方式是将这些字符都映射到一个足够大的足以包容它们两者的字符集里，例如Unicode，并用wchar_t处理这个字符集，写出对应代码。另一种方式是分别对各种字符类型和字符串定义相关的类，例如定义Jchar、Jstring、Hchar、Hstring，并使这些类可以正确地相互作用。这些类都应该由一个公共模板生成。按照我的经验，这样两种方式都可以工作。但任何触动国际化和多种字符集的决定都会引起激烈争论，其情绪化的速度远远快于其他任何问题。

另一个问题是能否以及如何允许在C++程序正文里使用扩充的字符集，这个问题同样很不简单。当然我很希望在需要苹果、树、小船和岛的程序里用丹麦语的单词表述这些概念。允许在注释里使用æble, træ, båd和ø一点也不难，在注释里使用英语之外的单词也是很常见的情况。但是允许在标识符里使用扩充字符集就会带来更多的问题。从原则上说，我不喜欢在C或C++的程序里允许用丹麦文、日文或者朝鲜文写标识符。虽然这样做并不存在严重的技术问题。实际上，在Ken Thompson写的一个内部的C编译程序就允许将所有没有特殊意义的Unicode字符用在C的标识符里面。

我担心的是兼容性和理解问题。从技术上说兼容性是可以处理的。但无论如何，英语毕竟作为程序员间的一种公共语言扮演了很重要角色，我觉得在没有经过认真考虑之前就抛弃它将是很不明智的。对于大部分程序员来说，系统化地使用希伯莱文、中文、朝鲜文等将带来严重的理解障碍，即使是我的母语丹麦文，也会使一般水平的说英语的程序员感到非常头痛。

C++委员会至今还没有在这个论题上做出任何决定，但是我想它将必须做些事情，而任何可能的解决方案都会存在很大的争议。

第7章 关注和使用

某些语言的设计是为了解决一个问题，
另一些的设计则只是为了证明一个观点。

——Dennis M. Ritchie

C++的使用——编译程序——会议、书籍和杂志——工具和环境——学习C++的途径——用户和应用——商业竞争——C++的替代物——期望和看法

7.1 关注和使用的爆炸性增长

设计C++就是为了为用户服务。它不是为设计一个完美语言而进行的学术试验，也不是为它的开发者敛财的一种商业产品。这样，为了满足其目标，C++必须拥有自己的用户——它也确实有了：

日期	C++用户的估计数
1979年10月	1
1980年10月	16
1981年10月	38
1982年10月	85
1983年10月	?? + 2 (考虑非Cpre用户)
1984年10月	?? + 50 (考虑非Cpre用户)
1985年10月	500
1986年10月	2 000
1987年10月	4 000
1988年10月	15 000
1989年10月	50 000
1990年10月	150 000
1991年10月	400 000

换句话说，在这12年里，C++用户的人数大约每七个半月增加一倍。这些都是保守估计的数字，C++用户是很难计算的。首先，存在着像GNU G++和Cfront这样的实现，它们被发布到许多大学里，在那里不可能存在有意义的用户数记录。其次，许多公司——包括工具的提供商和最终用户——都把它们那里使用者的数目及其所做工作的种类看作是一种机密。当然我毕竟还是有许多朋友、同事、联系人以及编译的提供商，他们因为相信我而提供了数字，只要我能以负责任的态度使用这些数字。这就使我有可能估计C++用户的数目了。这些估计数据的建立方式是：取出用户反馈给我的数字或者基于个人经验做一些估计，而后去掉各个数的尾数，而后将它们相加，再去掉尾数。这些数字是当时做出来的，后来也没有以任何方式调整。为了支持上面关于这些数字有些保守的断言，我可以提出Borland，它是最大的独立C++编译系统提供商。在1991年10月，Borland向公众宣布它已经销售了500 000个编译系统。这个数字是可能的也是可信的，因为Borland是一个上市公司。

现在C++的用户数目已经上升到这样一种程度，使我再也没有合理的方法去估计它了。我

想，已经无法把C++用户的当前数目估计到以十万人计的最接近数字。公开的数据显示，到1992年底销售的C++编译系统已经超过了一百万个。

7.1.1 C++市场的缺位

对我来说，这些数字中最令人吃惊的是C++获得其早期用户并不是通过传统市场完成的（7.4节）。相反，各种形式的电子通讯技术在这里扮演了关键性的角色。在最初的几年里，大部分发布和技术支持都是通过电子邮件完成的。关于C++的新闻讨论组也是很早就由用户们自己建立了起来。对网络的密集使用使有关语言、技术和工具状态的信息能够广泛普及。在今天这些东西已经很平常了，但是在1981年它还是相对很新的东西。我怀疑C++是第一个利用了这种途径的主要语言。

后来，更常规的通讯形式和市场出现了。在AT&T发布了Cfront 1.0之后，一些转销商，特别值得提出的是爱尔兰John Carolan的Glockenspiel公司和它们在美国的分销商Oasys（后来变成Green Hills）在1986年开始做小的广告。当独立开发C++编译系统，如Oregon Software公司的C++编译系统和Zortech的C++编译系统出现以后，C++就变成广告上的一个常见符号了（大约从1988年开始）。

7.1.2 会议

在1987年，USENIX（UNIX用户协会）的David Yost接到一个动议，要求召开一次专门讨论C++的会议。因为David不大肯定是否会有足够多的人感兴趣，这个会议被称为一个“专题讨论会”，David还在私下告诉我，“如果没有足够的人登记参加，它就将被取消”。他没有告诉我“足够的人”到底是什么意思，但我猜想了一个数字，大约在30人左右。David Yost选择来自国家卫生署的Keith Gorlen作为程序主席。Keith与我和其他人联系，收集我们听说过的有关项目的电子邮件地址，以便发送征集论文的电子邮件。最后接受了30篇论文，214人出现在新墨西哥州的圣菲市，这是1987年11月。

圣菲会议为后来的会议建立了一个很好的范例，文章中包括应用、程序设计和教学技术、改进语言的想法、库以及实现技术等。对于一个USENIX会议，很值得注意的是在这之中有关于在Apple Macintosh、OS/2、连接计算机上C++实现的文章，关于实现非UNIX操作系统（如CLAM [Call, 1987] 和Choices [Campbell, 1987]）的文章。NIH库 [Gorlen, 1987] 和Interview库 [Linton, 1987] 也在圣菲初次登台亮相。一个后来成为Cfront 2.0的早期版本也在那里展示，我第一次公开演示了它的各种特征 [Stroustrup, 1987c]。USENIX的C++会议一直是一个学术性和技术性的C++会议。出自这些会议的论文集是有关C++及其使用的最佳阅读材料。

如前所述，圣菲会议被定位为一个专题讨论会。由于其讨论的集中程度也使它实际上成为一个专题讨论会，虽然有200多参加者。无庸置疑，在下一次会议上专家们就被新人和那些想来弄清楚C++到底使什么的人们淹没了。这也使深入和技术性的讨论几乎无法进行，讲座和商业活动已经变成了压倒性的因素。在Andrew Koenig的建议下，一个“实现者专题研讨会”紧接着1988年在丹佛的USENIX C++会议之后召开了。在大会之后，几十个在会上发言的人、C++实现工作者等从丹佛出发，到Estes公园参加了一天热闹的讨论。特别是有关static成员函数（13.4节）和const成员函数（13.3节）的思想被很好地接受了，我决定将这些特征作为

Cfront 2.0的一部分。这件事由于AT&T的内部政策而稍微推迟了一点（3.3.4节）。在我的鼓动下，Mike Miller公布了他的一篇论文 [Miller, 1988]，这引起了将异常处理机制引进C++的第一次严肃讨论。

除了USENIX C++会议之外，还有许多针对C++、针对C包括C++、针对面向对象的程序设计包括C++的商业性的和半商业性的会议。欧洲的C和C++用户协会（ACCU）也安排了一些会议。

7.1.3 杂志和书籍

在1992年中，仅仅英语的有关C++的书籍就已经超过了100本，中文、丹麦文、法文、德文、意大利文、日文、俄文等翻译的和自己写出的书也有了许多。其质量差别当然也是非常大的。我也很高兴地看到那时自己的书已经被翻译成十来种语言。

第一个针对C++的杂志，*The C++ Report*，在1988年1月开始出版，Rob Murry是它的编辑。一个更大的印刷也更好的季刊，*The C++ Journal*在1991年春天出版，Livleen Singh是它的编辑。此外还有几个由C++工具提供商控制的新闻快报。许多专业杂志，如*Computer Languages*，*The Journal of Object-Oriented Programming*（JOOP），*Dr. Dobbs Journal*，*The C Users' Journal*和*.EXE*上也都有关于C++的专栏。Andrew Koenig在JOOP上的专栏在质量和不做宣传方面特别具有一贯性。讨论有关C++问题的这些出版物和它们的编辑政策也都变化得非常快。我提出这些杂志、会议、编译系统、工具等的目的，并不是想给出一个目前的“客户综述”，而只是想说明一下早期C++社团的宽广程度。

新闻组和公告板，例如usenet上的comp.lang.c++和BIX上的c.plus.plus，在这些年也产生了数以万计的消息，或是去取悦它们的读者，或是使他们失望。要想读完人们写出的所有有关C++的东西，在今天花掉全部时间也不行了。

7.1.4 编译程序

圣菲会议（7.1.2节）实际上也是C++实现的第二次浪潮的宣言。Steve Dewhurst描述了他和其他人一起为AT&T的Summit系列计算机实现的C++编译系统的体系结构。Mike Ball展示了后来变成TauMetric C++编译系统（更广为人知的名字是Oregon Software C++编译系统）的一些思想，这个编译程序是他和Steve Clamage在圣迭戈写的。Mike Tiemann给了一个最激动人的最有趣的报告，展示了他正在做的GNU C++如何能完成几乎所有事情，使其他所有的C++编译程序开发者都出了局。新的AT&T C++编译系统根本就没有实现，GNU C++的版本1.13是在1987年12月第一次发布的，而TauMetric C++在1988年1月交货。

直到1988年7月，所有在PC机上的C++编译还都是Cfront的移植。而后Zortech开始推出自己的编译系统，由Walter Bright在西雅图开发。Zortech编译系统的出现第一次使C++对于PC上的人们成为一种“真正的”东西。更多保守的人们保留着他们的评价权，直到1990年5月Borland发布了它的C++编译系统，甚至直到1992年3月Microsoft的C++编译系统出现。DEC在1992年1月第一次推出了自己独立开发的C++编译系统，IBM在1992年3月发布了它独立开发的第一个C++编译系统。现在已经有了十几个独立开发的C++编译系统。

除了这些编译系统外，Cfront几乎被移植到所有的地方，特别是Sun，Hewlett-Packard，Centerline，ParcPlace，Glockenspiel，Comeau Computing在它们几乎每个平台上都发送了基

于Cfront的产品。

7.1.5 工具和环境

C++的设计是作为一种在缺乏工具的环境里就能够生存的语言。部分地说这也是必须的，因为在早年间资源几乎是完全缺乏，后来则是相对贫乏。这也是一种有意的决定，以便能允许简单的实现，特别是允许实现的简单移植。

能与常规的支持其他面向程序语言的环境相匹敌的C++程序设计环境也已经开始出现了。例如，ParcPlace的ObjectWorks基本上是一个Smalltalk程序开发环境，依据C++修改而成。Ceterline C++（以前的Seber C++）是一个基于解释器的C++环境，是从Interlisp环境那里得到的灵感。这些也给了C++程序员一种机会，使他们能用到更快更昂贵，而通常也能有更高生产效率的环境。这些在以前只可以对其他语言使用，或者只是一种研究性的玩具，或者同时两者都是。环境是一种框架，使各种工具可以在其中互相操作。现在已经存在一大批这样的C++环境。在PC上的大部分C++实现都是嵌入在一个由编辑器、工具、文件系统和标准库等组成的框架里的编译程序。MacApp和Mac MPW是Apple Mac上的这种东西，ET++是具有MacApp风格的一个公共域（public domain）版本。Lucid的Energize和Hewlett-Packard的Softbench是另外的例子。

虽然这些环境已经复杂到超出了一般对C所用的东西，但也还只是更高级得多的系统的初步预演。写得很好的C++程序是一个等着使用的大信息容器。当前的工具不过是倾向于关注语言的语法方面、执行的运行时特征以及程序的行文观点等。为了挖掘出C++语言里的所有宝藏，程序设计环境必须超越静态程序所表现的简单文件和字符观点，能够理解和使用整个类型系统。它必须能以某种一致的方式将程序的运行信息与它的静态结构关联起来。自然，这样一个环境必须具有可伸缩性，以便能处理大的程序（例如500 000行的C++），在那种地方工具的作用就极端重要了。

一些这样的系统正在开发之中。我以个人方式深入涉足了一个这种系统 [Murry, 1992] [Koenig, 1992]。我认为也应该适时地提出一个警告。程序设计环境也可以被提供商用于将用户锁定在一个由特征、库、工具和工作模板封闭的世界里，不能很容易地转到其他系统去。这样，用户将会变得完全依赖于某一个特定的提供商，从而被剥夺了去使用该提供商不愿意支持的机器结构、库、数据库等的权利。我设计C++的主要目标之一就是给用户一种选择不同系统的权力，程序设计环境可以设计成与这个目标和谐的东西，当然它也不必是这样的 [Stroustrup, 1987b]：

“必须特别当心，应该保证程序源代码能够以很低的代价在不同的这类环境之间转移。”

与此同时，我看不会有单一的宏大的标准库，也没有希望出现一个单一的标准的C++软件开发环境 [Stroustrup, 1987d]：

“至少对于C++，必然同时存在几个不同的开发和执行环境，在这些环境之间必然有根本性的差异。期望在例如Intel 80286和Cray XMP上有同样的执行环境显然是不现实的。期望个人程序员和进行着大规模开发的200人程序组使用相同的程序开发环境同样也是不现实的。但无论如何，事情也很清楚，存在许多能够用来同时提升这两类环境的技术，因此人们必须努

力去发掘其中的共性，在所有有意义的地方。”

库、运行环境和开发环境的多样性是支持范围广泛的C++应用的最关键因素。这个观点至迟在1987年就已经指导着C++的设计，事实上它还更久远些。这个观点植根于把C++看成一种通用的程序设计语言（1.1节，4.2节）。

7.2 C++的教与学

C++使用的成长和性质也已经受到学习C++方式的强有力影响。由此可知，如果对C++应该如何教和学缺乏认识的话，要想理解它就可能比较困难。如果缺乏这种认识，可能就无法理解C++的某些快速成长的方面。

关于应该如何有效地将C++教给相对而言的新手，如何使他们能有效使用学到的东西，这些问题从很早就影响着C++的设计。我做了许多教学工作——至少是做了许多教授某些研究者（并不是专业的教育工作者）的工作。我在设法将自己的想法传播出去，在观察自己和别人所教的人们写出的实际程序方面有许多成功与失败，这些都对C++的设计产生了重要影响。

几年之后逐渐呈现出了一种方式：首先强调一些概念，随后再强调概念之间的关系和主要的语言特征。把各个单独语言特征的细节先放下，直到人们需要知道它们的时候再去学习。在这种方式不行的那些地方就修改语言本身去支持它。这种相互作用使语言逐渐成长为一种适宜用于设计的好工具。

与我一同工作的人和我教的那些人基本上都是职业的程序员和设计师，他们需要在工作中学习，不可能拿出几个星期或者几个月去学习新的技术。从这里也发展出设计C++的许多思想，以便使人能逐步学习它，逐步采纳它的特征。C++的组织方式使你能够以大致线性的方式学习它的概念，并在一路上获得实际利益。更重要的是，你的获益大致上可以正比于所花的精力。

我认为，有关程序设计语言、语言特征、程序设计风格等的许多讨论，其实际关注的更多的是程序设计语言特征的教育，而不是这些特征本身。对许多人而言最关键的问题是：

我没有多少时间去学习新技术和新概念，在这种情况下我怎样才能有效地开始使用C++？

如果某种语言对这个问题的回答比C++的更令人满意，人们就会选择那种语言，因为这个程度的程序员通常都有选择权（他们也应该选择）。1993年初我在compl.lang.c++里对这个问题给出了下面的回答：

“很清楚，使用C++‘最好’是在你对许多概念和技术都有了深入理解，达到了某种随心所欲的状态之后，但要想达到这种状态，你需要通过若干年的学习和实践。下面的方式通常是行不通的：告诉一个新手（C++的新手，而不是一般意义上的程序设计新手），首先取得对C、Smalltalk、CLOS、Pascal、ML、Eiffel、汇编语言、高性能系统、OODBMS、程序验证技术等的透彻理解，而后在他或她的下一个项目中把这些学过的课程应用到C++上。所有这些课题都值得学习，而且——从长远的观点看——确实也会有帮助，但是实际的程序员（和学生）不可能从他们正身陷其中的事情里抽出若干年的时间，去做对于程序设计语言和技术的深入学习和研究。

另一方面，大部分新手都知道‘一知半解是很危险的事情’，希望能得到一种保证：他们在开始自己的下一个项目之前/之中能够拿出时间来学的那一点点东西将确实能够帮助，而不是扰乱或者阻碍那个项目取得成功。他们也希望能够确信，这一点点能够立即吸取的新东西可以成为一条坦途的一部分，这条路能够指导他们走向他们真正期望的完整理解，而不是某些孤立技巧，向前走什么地方也到不了。

很自然，满足这些准则的途径肯定不止一条，究竟选择哪一条要看个人的背景，当前的需要以及可能花的时间。我认为许多教育培训工作者和在网络上发表意见的人都低估了这个问题的重要性：总的来说，‘教育’一大批人而去特别关心其中个别的人，从代价上看起来将会有有效得多——也更容易些。

下面考虑几个共性的问题：

我对C或C++都不了解，我是不是应该先学习C？

我想做OOP，那么是不是应该在学习C++之前先学Smalltalk？

我在一开始应该把C++作为一种OOPL，还是作为一个更好的C？

学习C++需要花多少时间？

我不是想说自己有对这些问题的惟一答案。正如前面所言，‘正确的’回答依赖于环境情况，大部分C++教科书的作者、教师和程序员都有他们自己的回答。我的回答是基于自己多年用C++和系统程序语言做程序设计，教授短的C++设计和编程课程（主要是给职业程序员），做C++入门和C++使用的咨询，讨论C++语言，以及自己关于编程、设计和C++语言的一般性思考。

我对C或C++都不了解，我是不是应该先学习C？不，首先学习C++。C++的C子集对于C/C++的新手是比较容易学的，又比C本身容易使用。原因是C++（通过强类型检查）提供了比C更好的保证。进一步说，C++还提供许多小特征，例如运算符new，与C语言对应的东西相比，它们的写法更方便，也更不容易出错。这样，如果你计划学习C和C++（而不仅仅是C++），你不应该经由C那条迂回的路径。为了能很好地使用C，你需要知道许多窍门和技术，这些东西在C++里的任何地方都不像它们在C里那么重要、那么常用。好的C教科书倾向于（也很合理）强调那些你将来在用C做完整的大项目时所需要的各种技术。好的C++教科书则不太一样，强调能引导你去做数据抽象、面向对象的程序设计的技术和特征。理解了C++的各种结构，而后学习它们在（更低级的）C里的替代物将是很简单的（如果需要的话）。

要说我的喜好：要学习C就用[Kernighan, 1988]；要学C++就用[2nd]。两本书的优点是都组合了两方面内容，一方面是关于语言特征和技术的指导性的描述，另一方面是一部完整的参考手册。两者描述的都是各自的语言而不是特定的实现，也不企图去描述与特定实现一起发布的特殊程序库。

现在有许多很好的教科书和许多各种各样风格的材料，上面只是我对理解有关概念和风格的喜好。请仔细选择至少两个信息来源，以弥补可能的片面性甚至缺陷，这样做永远是一种明智之举。

我想做OOP，那么是不是应该在学习C++之前先学Smalltalk？不。如果你计划用C++，那就学C++。各种语言，像C++、Smalltalk、Simula、CLOS和Eiffel等，各有自己对于抽象和继承等关键概念的观点，各语言以略微不同的方式支持着这些概念，也支持不同的设计概念。

学习Smalltalk当然能教给你许多有价值的东西，但它不能教给你如何在C++里写程序。实际上，除非你有充分时间学习和消化Smalltalk以及C++的概念和技术，否则用Smalltalk作为学习工具将导致拙劣的C++设计。

当然，如果同时学了C++和Smalltalk，使你能取得更广泛领域中的经验和实例，那当然是最理想的。但是那些不可能花足够时间去消化所有新概念的人们常常最后是‘在C++里写Smalltalk’，也就是说去用那些并不能很好地适应C++的Smalltalk设计概念。这样写出的程序就像在C++里写C或Fortran一样，远不是最好的东西。

常见的关于学习Smalltalk的理由是它‘很纯’，因此会强迫人们去按照‘面向对象的’方式思考和编程。我不想深入地讨论‘纯’的问题，除了提一下之外。我认为一个通用的程序设计语言应该而且也能够支持一种以上的程序设计风格（范型）。

这里的问题是，适合Smalltalk并得到它很好支持的风格并不一定适合于C++。特别是模仿性地追随Smalltalk风格，将会在C++里产生低效、丑陋且难以维护的C++程序。个中的理由很简单，好的C++程序所需要的设计应该能很好地借助于C++静态类型系统的优势，而不是与之斗争。Smalltalk（只）支持动态类型系统，把这种观点翻译到C++将导致广泛的不安全性和难看的强制转换。

我把C++程序里的大部分强制转换看成是设计拙劣的标志。有些强制转换是很基本的，但大部分都不是。按照我的经验，传统C程序员使用C++，通过Smalltalk理解OOP的C++程序员是使用强制转换最多的人，而所用的那些种类的转换完全可以通过更仔细的设计而得以避免。

进一步说，Smalltalk鼓励人们把继承看成是惟一的，或者至少是最基本的程序组织方式，并鼓励人们把类组织到只有一个根的层次结构中。在C++里，类就是类型，并不是组织程序的惟一方式。特别地，模板是表示容器类的最基本方法。

我也极端怀疑一种论断，说是需要强迫人们去采用面向对象的风格写程序。如果人们不想去学，你就不可能在合理的时间内教会他们。按照我的经验，确实愿意学习的人从来也不短缺，最好还是把时间和精力用到他们身上。除非你能把握住如何表现隐藏在数据抽象和面向对象的程序设计后面的原理，否则你能做的不过是错误地使用支持这些概念的语言特征，而且是以一种不适当的‘巴罗克’形式[⊖]——无论在C++、Smalltalk或者其他语言里。

参看《C++程序设计语言（第2版）》[2nd]，特别是第12章，那里对于C++语言特征和设计之间关系有更多的讨论。

我在一开始应该把C++作为一种OOPL，还是作为一个更好的C语言？看情况。为什么你想开始用C++？对这个问题的回答应该能确定你走近C++的方式，在这里没有某种放之四海而皆准的道理。按照我的经验，最安全的方式是自下而上地学习C++，也就是说，首先学习C++所提供的传统的过程性程序设计特征，那个更好的C子集；而后学着去使用和遵循那些数据抽象特征；再往后学习使用类分层去组织相互有关的类的集合。

按照我的观点，过快地通过早期阶段是很危险的，这样就会使忽视某些重要概念的可能性变得非常之大。

例如，一个有经验的C程序员可能会认为C[⊖]的更好的C子集是‘很熟悉的’，因此跳过了

⊖ 意为：花哨的形式。——译者注

⊖ 原书如此，疑为C++之误。——译者注

教科书中描述这个方面的前100页或多少页。但在这样做时，这个C程序员可能就没看到有关函数的重载能力，有关初始化和赋值之间差异的解释，用运算符new做存储分配，关于引用的解释，或许还有其他一些小特征。在后面的阶段它们会不断地跳出来缠住你，而在这时，一些真正的新概念正在复杂的问题中发挥着作用。如果在更好的C中所使用的概念都是已知的，读过这100页可能也就只要几个小时的时间，其中的一些细节又是有趣的，很有用的。如果没有读，后面花的时间就可能很多了。

有些人表达了一种担心，害怕这种‘逐步方式’会引导人们永远去写C语言风格的东西。这当然是一种可能的后果，但是从百分比看，也很难说这样做就一定比在教学中采用‘更纯的’语言或者强迫的方式更不值得信任。关键是应该认识到，要把C++很好地用作数据抽象和/或面向对象的语言时应该理解几个新概念，而它们与C或者Pascal一类语言并不是针锋相对的。

C++并不只是用新语法表述一些老概念——至少对于大部分程序员而言不是这样。这也就隐含着教育的需要，而不仅仅是训练。新的概念需要通过实践去学习和掌握。老的反复试验过的工作习惯需要重新评价。不再是按照‘老传统的方式’向前冲，而是必须考虑新的方式——通常在以新方式做事情，特别是第一次这样做时，与按照老方式相比一定更困难，也更花费时间。

许许多多的经验说明，对于大部分程序员而言，花时间和精力去学习关键性的数据抽象和面向对象技术都很有价值。并不是非经过很长的时间才能产生效益，一般在3到12个月就可以。不花这些精力而只是使用C++也会有效益，但最大的效益还是要在为学习新概念而花费精力之后——我的疑问是，如果什么人不想花这个精力，那么为什么还要转到C++来呢。

在第一次接触C++，或是在许多时间之后又第一次接触它，用一点时间去读一本好的教科书，或者几篇经过很好选择的文章（在*The C++ Report*和*The C++ Journal*里有许多这样的文章）。你也可能想看看某些主要的库的定义和源代码，分析其中所使用的概念和技术。对那些已经用了一段C++的人来说，这也是个好主意，在重温这些概念和技术的过程中可以做许多事情。自C++第一次出现以来，在C++语言以及与之相关的编程和设计技术方面已经发生过许许多多事情。将《C++程序设计语言》的第1版和第2版做一个简单对比，就足以使人相信这个说法。

学习C++需要花多少时间？同样要看情况。依赖于你的经验，也依赖于你所说的‘学习C++’的意思。对于大部分程序员来说，学习语法和用更好的C的风格写C++，再加上定义和使用几个简单的类，只要一两周时间。这是最容易的部分。最主要的困难在于掌握新的定义和编程技术，这也是最有意思最有收获的部分。曾经和我讨论过的大部分有经验的程序员说，他们用了半年到一年半时间，才真正觉得对C++适应了，掌握了它所支持的数据抽象和面向对象技术。这里假定他们是在工作中学习并维持着生产——通常在此期间用C++的某种‘不那么大胆’的风格做程序设计。如果你能够拿出全部时间学C++，你就可能更快地适应它。但是，在没有将新思想设计应用到真实的项目中之前，这个适应也很可能是骗人的。面向对象的编程和面向对象的设计基本上是实践性的训练而不是理论训练。只是对一些玩具式的小例子使用或者不使用它，这些思想就很可能演化为一种危险的‘宗教’。

请注意，学习C++，最根本的是学习编程和设计技术而不是语言的细节。在做完了一本教

科书的学习工作之后，我会建议一本有关设计的书，例如 [Booch, 1991][⊖]，这本书里有一些稍长的例子，用的是5种语言（Ada, CLOS, C++, Smalltalk和Object Pascal），这样就可能在某种程度上避免语言的偏狭性，这种偏狭性弄糟了许多有关设计的讨论。在这本书里我最喜欢的部分就是描述设计概念和例子的那几章。

关注设计方式与非常仔细地关注C++定义细节的方式（例如ARM，其中包含许多有用的信息，但是没有关于如何用C++编程的信息）是截然不同的。把注意力集中到细节上很容易把人搞得头昏脑涨，以至于根本就用不好语言。你大概不会试着从字典和语法去学习一种外国语吧？

在学习C++时，最根本的应该是牢记关键性的设计概念，使自己不在语言的技术细节中迷失了方向。如果能做到这一点，学习和使用C++就会是非常有趣的和收效显著的。与C比较，用一点点C++就可能带来许多收获。在理解数据抽象和面向对象技术方面付出进一步努力将能得到更多的收获。”

这个观点也不是全面的，受到当前工具和库状况的影响。如果有了保护性更强的环境（例如包括了广泛的自动的运行时检查）以及一个定义良好的基础库，你就可以更早地转到大胆使用C++的方面去。这些将能更好地支持从关注C++的语言特征到关注C++所支持的设计和编程技术的大转移。

分散一些兴趣放到语法上，把一些时间用到语言的技术细节上也是非常重要的。有些老牌程序员喜欢翻弄这些细节。这样的兴趣与不大情愿去学习新的程序设计技术通常是很能分辨清楚的。

类似地，在每个课程和每个项目里总有这样的人，他们根本不相信C++的特征是可以负担得起的，因此在后来的工作中坚持使用自己更熟悉和信任的C子集。只有不多的有关个别C++特征和用C++写出的系统的执行效率方面的数据（例如，[Russo, 1988]、[Russo, 1990]、[Keffer, 1992]），就说比C更方便的机制是不可能负担的，这样做就想抑制住那种已经牢牢建立起来的观点是几乎无望的。看到这种宣传的量和语言或工具领域中未得到满足的允诺的数目，人们应该抱着怀疑态度，并要求更明显的证据。

在每个课程和项目里也总有另一种人，他们确信效率无关紧要，倾向于用更一般的方式去设计系统，结果是即使在最先进的硬件上也产生了可观察到的延迟。不幸的是这种延迟在人们学习C++的过程中写玩具程序时是很难观察到的，因此具有这种性质的问题常常被遗留下来，直到遇到真正的项目。我一直在寻找一个简单但又实际的问题，如果采用一种过分一般的方式去解决，它就能打倒一个很好的工作站。这样的问题将使我能证明带有倾向性的设计的价值，以抵制那些极端的乐观主义者；而通过仔细思考又能使性能大大改善，可以对付过分谨慎和保守主义者。

7.3 用户和应用

我关于C++已经被用于做了些什么，以及还可能被使用到的其他地方的观察也一直在影响着C++的发展。C++特征的增长基本上就是为了响应这些真实的或者设想中的需要。

[⊖] Booch书的第2版 [Booch, 1993] 自始至终使用的是C++。

在C++的使用中有一个方面已经反复在我的头脑里引起重视：以某种方式看，C++应用在数量上的失调是很古怪的。这当然可能反映出更有意义的是去讨论那些不寻常的项目，但我还是怀疑其中存在着某种更根本的原因。C++的强项是它的灵活性、效率和可移植性。这就使它成为某些项目的强有力候选者，如果在项目中涉及到各种不寻常硬件、不寻常的操作环境或者存在与几种语言的接口等。这种项目的一个例子是华尔街系统，它需要在主机上运行并与COBOL代码相互操作；在工作站上与Fortran代码相互操作；在PC上与C代码相互操作；还要在网络上和所有这些东西连接。

我认为这反映出C++已经站在工业产品代码的前沿上了。在这里，C++的着眼点与那些特别倾向于试验性应用的语言不同——无论这个应用是工业的、学术的或者教育的。C++除了在教育中使用外，也已经很自然地被广泛应用到试验性和探索性工作中。无论如何，它在产品代码中所扮演的角色已经成为设计决策中的决定性因素。

7.3.1 早期用户

带类的C和C++的早期世界很小，带有高度的个人接触色彩，这使人们能透彻地交换思想并迅速得到对问题的反馈。这样，我在那时就能够直接检验用户的问题，用排除错误之后的Cfront或者库，偶尔也用语言的修改作为对有关问题的响应。正如在2.14节和3.3.4节中所说，这些用户大部分是贝尔实验室里的研究和开发工作者，当然也有些例外。

7.3.2 后来的用户

不幸的是，许多用户并不操心去记录下他们的经验。更糟糕的是，许多机构把试验数据作为高度机密。因此就产生了许许多多关于程序语言和编程技术的神话和误报——有时甚至根本就是胡扯八道。这些东西至少与真实的数据一样多，吸引着程序员和管理者的注意力。这样就导致重复花费了许多精力，重复地犯了许多已知的错误。本节的目的是想给出一些已经使用了C++的领域，并鼓动开发工作者们能够以某种方式写出他们的工作，以使整个C++社团从中获益。我希望这能给人们一种印象：广泛应用一直在影响着C++的成长。这里提到的每个领域都代表了至少两个人两年的努力。我看到过文档的最大项目包含了500万行C++代码，由200个人经过7年开发和维护。

动画、自动潜水艇、收费系统（电信）、保龄球道控制、线路路由选择（电信）、CAD/CAM、化学工程过程模拟、汽车分销管理、CASE、编译系统、控制台软件、回旋加速器模拟和数据处理、数据库系统、排错系统、决策支持系统、数字照片处理、数字信号处理、电子邮件、刺绣机控制、专家系统、工厂自动化、财务报告、空战遥测、外币兑换处理（银行）、资金转账（银行）、家谱搜索软件、加油站抽油控制和收费、图形学、硬件描述、医院记录管理、工业机器人控制、指令集模拟、交互式多媒体、磁流体动力学、医疗图像、医疗监控、抵押公司管理（银行）、网络、网络管理和维护系统（长途电话）、网络监控（长途电话）、操作系统（实时、分布式、工作站、主机、“完全面向对象的”）、程序设计环境、养老金（保险）、激波物理模拟、屠宰场管理、SLR照相机^Θ软件、开关软件、检测工具、贸易系统（银行）、交易处理、传输系统（长途电话）、运输系统的车队管理、用户界面、电子游戏、

^Θ SLR, Single-Lens-Reflex, 单镜头反射式照相机，俗称“单反相机”。——译者注

以及虚拟现实等。

7.4 商业竞争

商业竞争者基本上没有受到重视，C++语言按照它自己的计划、它自己的内部逻辑和用户的经验发展起来。在程序员之间过去有现在也一直有许多讨论，在出版物上、在会议中、在电子公告板里，讨论的是哪种语言“最好”，哪种语言能在某类竞争中“赢得”用户。按我个人的观点，我认为争论中的许多东西都是误入了歧途或者是误传。但这些问题也是很现实的，对那些需要在自己的下一个项目中选择程序设计语言的程序员、管理者或者教授而言都是如此。无论是好是坏，人们总带着一种几乎是宗教式的热情去参加有关程序设计语言的争论，通常都认为程序语言的选择是对一个项目或者组织的最重要选择。

理想情况是人们应该针对每个项目选择最好的语言，因此在一年之中使用许多语言。在现实中大部分人没时间去学习一种新语言，即使是在某领域里该语言是很有效的工具，能增进多种语言的专业技能。因为这种状况，为个别的程序员或者组织评价一种程序语言也成了挑战性的工作，很少能做好——更少有人能写出对其他人也有用的不带偏见的文件。此外，各种组织（为了好的或者坏的理由）都发现管理混合语言的软件开发特别困难。这种问题又由于语言设计者和实现者的行为方式而变得更加严重，因为这些人通常都不认为自己的语言与其他语言代码之间的合作有什么重要性。

事情还在进一步恶化，因为实际上程序员需要把程序语言作为一种工具来评价，而不是简单地作为一种智力产品。这就意味着程序员将关注实现、工具、各种形式的性能、支持组织、库、教育方面的支持（如书籍、杂志、会议、教师、顾问）等，根据他们当前的情况和他们可信的短期开发项目。考虑长远常常是太冒险了，由于存在着如泛滥洪水似的商业宣传和不切实际的幻想。

早年里许多人认为Modula-2是C++的竞争者。当然在1985年C++的商业版本出现之前，它还很难被认为是任何语言的竞争者。而到了那时，我觉得在美国Modula-2早已经被C语言大大地超过了。后来又出现了一种流行的疑问：到底C++和Object C中哪个才是面向对象的C。Ada也常常被许多原来可能选择C++的组织选中。进一步说，Smalltalk [Goldberg, 1983] 以及Lisp的某些面向对象的变形 [Kiczales, 1992] 也经常被用于一些应用，只要那里不要求赤裸裸的系统工作和最高的性能。后来有人针对一些应用把C++与Eiffel [Meyer, 1988] 和Modula-3 [Nelson, 1991] 做过比较。

7.4.1 传统语言

我个人的观点与这些都不同，我认为C++最主要的竞争对手还是C。究其原因，C++是今天使用最广泛的面向对象语言，它过去是现在也还是惟一的一个在C语言的地盘里能够与C媲美的语言——而与此同时又有一些重要的改进。C++提供了一条途径，使人可以从C语言转到另一种系统设计风格，基于应用层概念到语言概念的更直接映射（通常被称为数据抽象或面向对象的程序设计）完成系统的实现。其次，许多需要考虑新程序设计语言的组织都有一种传统，愿意采用某种家酿的语言（常常是某种Pascal变形）或者Fortran。除了在繁重的科学计算领域之外，与C++相比，这些语言都被认为是大致上与C等价。

我深深地钦佩C语言的实力，虽然大部分语言专家可能不赞成我的意见。按我的观点，他

们过于被C的明显瑕疵蒙住了眼睛，以至没有看到它的真正实力（2.7节）。我对付C的策略很简单：做所有C能够做的事情，在各个方面和每个地方都能做得与C一样好甚至更好一点；此外再为实际程序员提供某些C无法提供的服务。

Fortran是很难超过的。它有一些献身者，他们就像许多C程序员一样并不大关心程序设计语言或者计算机科学的更完美的观点，简单地就是想做完自己的事情。这常常也是一种合理的态度，因为他们智力上的兴趣在其他地方。许多Fortran编译系统在为某些高性能机器生成高效代码方面是极其卓越的，这常常也是Fortran用户最关心的事情。究其原因，部分地是因为Fortran有很松的防止别名规则；部分是因为在某些机器上关键性的数学子程序常规地被在线使用，而且真有价值；还由于人们花在Fortran编译系统上的精力和智慧。C++在与Fortran的竞争中很少取得成功，它极少被用到高性能的数学和工程计算里。但是这种情况还是会有的，C++编译系统在某些领域中正在变得更成熟、更有进取心，例如在在线化的领域里。C++语言中也能直接使用Fortran成熟的程序库。

C++正在越来越多地被用到数值和数学计算中 [Forslund, 1990] [Budge, 1992][Barton, 1994]。这也导致了一些扩充建议。一般地说，这些建议都受到Fortran的影响，并不是很成功。这也反应出我们的一种愿望：更多地关注高级的抽象机制而不是特定的语言特征。我的希望是：更多地关注高层次的特征和优化技术，从长远的观点看，这样做能更好地为科学和数值计算的社团的人们服务，比简单地加进几个低层次的Fortran特征更好。我也把C++看成是一种科学计算的语言，也希望对这种工作的支持能够比今天做得更好。实际的问题不是“要不要？”而是“怎么办？”。

7.4.2 更新一些的语言

第二个竞争是在C++和其他支持抽象机制的语言（也就是那些面向对象的语言和支持数据抽象的语言）之间进行的。在早些年（1984~1989年），如果考虑市场的话，C++一直处于下风。特别是因为AT&T在这个阶段的市场投入常常是零，花在C++上的广告费一共才3000美元，其中1000美元用于给UNIX用户发平信，通知他们C++的存在和正在销售。这看起来没起任何作用。另外2000美元花在接待1987年圣菲第一次C++会议的出席者们。这对于C++也没有多大帮助，但是至少我们都喜欢那个宴会。在第一次OOPSLA会议上，AT&T的C++人只能付得起所提供的最小房间。在这个房间挤满了志愿者，用黑板作为能够负担得起的计算机的替代品，用招贴纸复制技术论文代替精致的讲义。我们想做一些C++徽章，但是却无法找到钱。

直到现在，AT&T在C++舞台上最明显的行动就是遵循贝尔实验室的传统政策，鼓励开发和研究人员做报告、写论文、参加会议，它没有任何有意识地推动C++的政策。即使在AT&T内部C++也是一种草根运动，既没有钱也没有管理方面的压制。无论如何，贝尔实验室还是用来自AT&T的经费帮助了C++，但是生存在一个大公司的环境里，这种资助来得也很不容易。

在与新语言的竞争中，C++最根本的强项是它在传统环境（社会和计算机化的环境）中的活动能力、运行的时间和空间效率、它的类概念的灵活性、它的低价以及它的非专有权的特性。它的弱点是从C语言继承来的某些污浊成分，它缺少特定的新特征（例如内置的数据库支持），缺少特定的程序设计环境（直到后来，人们从Smalltalk或Lisp转移过来的一些这类环境才能对C++使用；参见7.1.5节），缺乏标准库（后来有了一些对C++广泛可用的主要的库——

但不是“标准的”，参见8.4节），而且它缺乏销售人员去平衡那些富有的竞争者们的努力。随着C++现在在市场上的主导地位，这最后一个因素已经消失了。毫无疑问，一定会有一些C++销售商使C++社团感到羞辱，他们会模拟某些商人和广告商为企图干扰C++的发展所使用的那些卑劣伎俩和无耻行为。

在与传统语言的竞争中，C++的继承机制是一个主要的加分因素。在与有继承机制的语言的竞争中，C++的静态类型检查又成为主要的加分因素。在所有上面提到的语言中，只有Modula 3和Eiffel以与C++类似的方式组合了这两方面特征。在Ada的修订版Ada 9X里也提供了继承。

C++是作为一种系统编程语言、作为一种为开发由大的系统部件组成的应用而进行设计的。这是我与我的朋友们非常熟悉的领域。我们不打算用C++在这个领域中最强的方面与拓宽其魅力做什么交换，这个决定对于C++的成功是至关紧要的。那么在这里是否曾也为迎合许多观众而损害到C++的能力呢，只有时间才能回答我们。我不会把这种情况看成是一个悲剧，因为我是这类人中的一个，我们都认为某个单一的语言不应该成为所有人的全部东西。C++已经很好地为它设计时考虑的那个社团提供了服务。我推测，通过库的设计C++的吸引力还能进一步扩大（9.3节）。

7.4.3 期望和看法

人们经常对AT&T容许其他人实现C++的行为表示很诧异，这实际上说明了他们很不了解法律以及AT&T目标。一旦出版了C++的参考手册 [Stroustrup, 1984]，就再没有办法去阻止任何人写出一个实现了。进一步说，AT&T不止是允许其他人加入C++的实现、工具和教育等这样一个蓬勃发展的大市场，它还欢迎和鼓励他们这样做。有一个大部分人都忽略了的事实，那就是AT&T作为程序设计产品的消费者远远大于它作为一个生产者。因此，AT&T从C++的领域里的“竞争者”那里获益匪浅。

AT&T当然希望C++成功，但没有一种公司语言能够在这样大的程度上获得成功。完美的实现、工具和教育的基础结构，这个代价实在太高了，任何公司都负担不起，无论它有多么大。一种公司语言必然会偏向于反应公司的政策和纲领，这些都会妨碍它在一个更大、更开放、更自由的世界里生存的能力。当然，如果任何语言能同时在两种环境中生存，一方面是贝尔实验室内部政策的压力，另一方面是一个更恶劣的开放市场，我很怀疑能够说它是个彻头彻尾的坏东西——即使它不像是遵循了学术潮流的旨意。

当然，非人格的公司并不是魔术式地生产政策，政策是由人制定的，是在许多人之间达成的一种共识。有关C++的政策来自于贝尔实验室计算机科学研究中心和AT&T其他地方所弥漫的一种思想。我在这种思想（在关系到C++时）的形成方面也很活跃，但是如果普遍可用软件的概念还没有被广泛接受的话，我也不可能有机会使C++成为能广泛使用的东西。

很显然，并不是每个人在所有时候都赞同这些。有人告诉我说，某管理层人士有一次就提出了明确的想法，要把C++的机密作为AT&T的一种“竞争优势”。他的想法后来被另一个管理者的一句话阻止了：“无论如何这个问题还得讨论，因为Bjarne已经向公司之外发送了700个参考手册的拷贝。”这些参考手册当然都是在正常的批准和我的管理层的鼓励之下发送的。

还有一个对C++既好也不好的重要因素，那就是使C++社团接受C++的许多不完美之处的

愿望。这种开放性对那些由于多年与软件工具工业的人和公司打交道，已经变得愤世嫉俗的人们是又一重的保证。但是它也激怒了那些完美主义者，成为对C++的公正的和不那么公正的批评的丰富源泉。公平地看，我认为在C++社团内部不断向C++投掷石块已经成为一个优点，它使我们保持诚信，促使我们不断地去改进语言及其工具，并保证C++的用户和潜在用户的期望能得以实现。

我讨论了“商业竞争”而没有提出任何特殊的语言特征、特殊的工具、发表的日期、市场策略、调查或者商业组织，对此有些人表示了诧异。部分地说这是被语言战争烤伤过的结果。在这种战争里，各种语言的辩护者们在争执中都带着宗教的狂热，以市场游说的方式说话，冷嘲热讽在那里占据了主导地位。在这些情况下，理智诚信和事实并不能获得额外的奖赏，而“争论”技术则只属于周围的极端政治派别。我确实是这样认为的。可悲的是，人们常常忘记，实际中永远存在对各种各样语言的需求，需要真正壁龛里的语言，也需要试验性的语言。称赞某种语言，比如说C++，并不意味着批评所有其他的语言。

更重要的是，我对于语言选择的讨论总是基于一种信念，这就是说，个别的语言特征和个别的工具在一个更大的场景中并没有那么重要，只能作为一种不那么科学的小争执中的关注点。多数法则的某种变形也在这里起着作用。

在这里提到过的所有语言都能完成项目中比较简单的那些部分，C语言也可以。在这里提到过的所有语言在做项目中的简单部分时都能比C做得更漂亮。通常这并不太重要。重要的是在一个长期的过程中，在一个语言里是否能把一个项目的所有部分都做好，是否能够用这个语言把一个机构（无论它是一个公司还是一个大学）在一段时间里遇到的所有主要项目都做好。

真正的竞争并不是个别语言特征美不美的竞赛，甚至不是语言完整规范的竞赛，而是关于它们的用户社团在所有的方面、所有的差异、所有的发明创造之间的竞争。由某个伟大思想统一起来的组织良好的用户社团具有某种局部的优越性，但是从长远看，从更广大的图景看，这种社团存在着致命的弱点。

优雅有可能在某种无法接受的代价下得到。“优雅”的语言将最终被抛弃，如果获得这种优雅所付出的是：限制了应用领域的代价，运行时间或者空间效率方面的沉重代价，限制了可以使用这个语言的系统的范围，技术代价与一个机构能够吸收的代价相比太不合适，过分地依赖于某个特定商业机构，等等。C++特征的广泛性，它的用户社团的多样性，它处理庸俗细节的能力等都是C++真正的利刃。C++具有与C语言相当的运行时间效率，而不是慢两倍、三倍或者十倍，这也是很有帮助的。

第8章 库

只有向后看才能理解生活，
但是人却只能向前生活。
——齐克果

库设计的折衷——库设计的目标——语言对库的支持——早期的C++库——I/O流库——并行支持——基础库——持续性和数据库——数值库——专用的库——一个标准C++库

8.1 引言

设计一个库比增加一个语言特征更好，这种情况是经常出现的。类可以表示我们需要的几乎所有概念。一般说库在语法方面起不了多少作用，但建构函数和运算符重载有时会有些用处。如果需要的话，也可以用C++之外的其他语言做函数的编码，以实现特殊的语义或者更好的性能。这方面的一个例子就是通过（在线的）运算符函数，将代码展开为向量处理硬件的代码，以这种方式提供高性能的库。

因为没有任何语言能够支持人们所需要的全部特征，又因为即使语言的扩充被接受也还需要时间去实现和部署，人们应该总把库作为第一选择。设计一个库，实际上经常能成为追求新机制的狂热的一种最具建设性的发泄方式。只有在迈向库的道路真正走不通的情况下，才应该踏上语言扩充之路。

8.2 C++库设计

一个Fortran库就是汇集起来的一些子程序；一个C库是汇集起来的一些函数，再加上若干与它们有关的数据结构；一个Smalltalk库则是植根于Smalltalk类层次结构中里一个层次结构。那么一个C++库又是什么呢？很清楚，一个C++库可以很像Fortran、C或Smalltalk库。它也可以是一组抽象类型，带着若干实现（13.2.2节）、一组模板（第15章），或者是一个混合体。你还可以进一步想出一些其他东西。C++库的设计者们对于库结构可以有多种选择，甚至可以为一个库提供多种不同的界面。例如，一个被组织为一组抽象数据类型的库，对于一个C程序则可以表现为一组函数；一个按照层次结构组织起来的库，对于客户而言又可能表现为一组句柄。

我们明显是面对着一个机会，但是我们能把握住结果的多样性吗？我认为我们可以。这种多样性正反映了C++社团的需要的多样性。一个支持高性能科学计算的库的约束条件当然与一个支持交互性图形的库大不相同，而它们两者的需要又与为其他库的构造者提供低级数据结构的库不同。

C++的发展已经能够包容这些具有多样性的库结构了，有些C++新特征的设计就是为了使各种库的共存变得更容易些。

8.2.1 库设计的折衷

早期C++库的设计显示出一种倾向，那就是去模仿能在其他语言里发现的库设计风格。例如我原始的作业库 [Stroustrup, 1980b] [Stroustrup, 1987b] ——最早的C++库——就提供了类似于Simula 67模拟机制的功能，复数算术库 [Rose, 1984] 提供的函数类似于在C语言数学库中为浮点运算提供的那些东西，而Keith Gorlen的NIH库则提供了Smalltalk库的一个C++仿制品。新一些的“早期库”仍然像是程序员由其他语言移植过来的，他们在提供库时还没有完整地吸收C++的设计技术，没有领会C++里各种可能的设计折衷。

在这里存在哪些折衷？在回答这个问题时程序员经常把注意力集中在语言特征上：我要不要使用在线函数？虚函数？多重继承？单根层次结构？抽象类？重载函数？这种关注点根本就是错的。这些语言特征的存在是为了支持更具本质性的折衷：设计是否应该

- 强调运行时的效率？
- 使修改之后的重新编译达到最小化？
- 最大化跨平台的可移植性？
- 允许用户扩展基本的库？
- 允许在没有源代码的情况下使用？
- 与现存的记法和风格混合到一起？
- 使之可以从不是用C++写的代码中调用？
- 对新手也容易使用？

给出了对于这些问题的回答之后，自然而然地也就有了对于语言层问题的答案。新型的库通常提供许多各种各样的类，以便使用户也可以做这类折衷。例如，一个库可能提供了一个非常简单而高效的字符串类，它可能还提供了另一个高级的字符串类，带有更多功能，为用户修改其行为提供更多的可能性（8.3节）。

8.2.2 语言特征和库的构造

C++的类概念和类型系统的一个最重要的考虑就是要支持C++库的设计。它的强弱直接确定了C++库的构形。我对库的建设者和使用者的建议非常简单：不要去与类型系统做斗争。与语言的最基础机制作对，即使能赢得胜利，其代价也必定是极其昂贵的。优雅、易于使用和效率只能在一个语言的基本框架内得到。如果这个框架对于你想做的事情不配合，那可能就说明你需要去考虑其他语言了。

C++的基本结构鼓励一种强类型风格的程序设计。在C++里，一个类就是一个类型。继承的规则、抽象类的机制与模板机制的组合，都是想鼓励用户严格依据它们为使用者提供的界面去操作各种对象。更直率地说出这个意思：不要用强制转换去打破类型系统。强制转换对于许多低级动作是必需的，偶尔也用来把高层东西映射到低级的界面上。但是，一个要求其最终用户广泛进行强制转换的库，实际上是给他们强加了一种过度而又不必要的负担。C语言的printf函数族、void*指针、联合以及其他低级机制都应该避免出现在库的界面上，因为它们蕴涵着类型系统的漏洞。

8.2.3 处理库的多样性

你不能直接取出两个库就假定它们能够在一起工作。许多库真的能这样，但是为了成功

地组合使用，一般说还是要考虑某些问题。有些问题应该由程序员去考虑，另一些则应该由库的建造者考虑，也有几个是语言设计者的事情。

许多年来，C++的发展一直是向着一个目标，那就是让语言提供充分的支持，以解决用户在试图使用两个独立设计的库时可能出现的各种基本问题。作为补充，库的提供商在设计自己的库时也应该开始考虑多个库的同时使用问题。

名字空间就是针对不同库里使用同样的名字而提供的机制（17.2节），异常处理为建立一种处理错误的公共模型提供了基础。模板（第15章）是为定义独立于具体类型的容器类和算法而提供的一种机制，其中的具体类型可以由用户或者其他库提供。建构函数和析构函数为对象的初始化和最后清理提供了一种公共模型（2.11节）。抽象类提供了一种机制，借助于它可以独立地定义界面，与实际被接口的类无关（13.2.2节）。运行时的类型信息是为寻回类型信息而提供的一种机制，因为当对象被传递给一个库再传递回来的时候，可能只携带着不够特殊的（基类的）类型信息（14.2.1节）。当然，这些不过是有关语言功能的一方面用途，但如果把它们看成是针对由独立开发的库出发去构造程序的支持功能，问题就更清楚了。

在这个方向上考虑多重继承（12.1节）：由Smalltalk获取灵感而开发的库通常都基于一个“通用的”根类。如果你有两个这样的东西，那就该你倒霉了。但是，如果库本来就是为两个不同应用领域写的，多重继承的最简单形式有时就会有所帮助：

```
class GDB_root {
    public GraphicsObject,
    public DataBaseObject {};
```

有一个问题在这里没有简单解决办法：如果在这两个“通用的”基类里提供了相同的服务。例如，两个库可能都提供了运行时的类型识别机制以及对象I/O机制。这类问题中有一些最好是通过分离出共同功能，放进标准库或者语言机制的方式解决。另外的则可以通过在一个新的公共根里提供功能的方式处理。当然，合并“通用的”库绝不会是件容易的事。最好的方式是库的提供商认识到他们并不拥有整个世界，将来也不会如此，而基于这种认识设计出的库实际上最符合他们本身的利益。

存储管理还给库的提供商提出了另外一些一般性的问题，特别是对那些使用了多个库的用户（10.7节）。

8.3 早期的库

用带类的C写出的最早的实际代码就是作业库 [Stroustrup, 1980b]（8.3.2节），它提供的为模拟而用的类似Simula的并行性。最早的一批真实程序是网络流量模拟、电路版布局等等，它们都使用了作业库。这个作业库今天仍然被大量使用着。从第一天开始C语言的标准库就可以在C++里使用——没有额外代价，也不需要重新编译。C语言的其他库也一样。经典的数据类型，例如字符串、检查边界的数组、表等，都是设计C++和测试其早期实现时所用的例子（2.14节）。

由于缺乏对参数化类型描述（9.2.3节）的支持，有关容器类（如链表和数组）的早期工作受到严重的阻碍。由于没有语言的支持，我们必须用宏来做这些东西。对C语言预处理器的宏功能，我们能说的最好的话就是它使我们能取得有关参数化类型的实践经验，也支持了个人和小工作组的使用。

在类设计方面的大部分工作都是与Jonathan Shopiro合作做的，他在1983年做出了表和字符串类，这些类在AT&T内部广泛使用，也是今天可以见到的，由贝尔实验室开发并由USL销售的“标准组件”库里的那些类的基础。这些早期库的设计都与语言设计有直接的相互影响，特别是与重载机制的设计。

早期的字符串和表库的关键目标是提供了一些相对简单的类，它们可以用作基本的构造块，用于实际应用或者更雄心勃勃的类。另一种典型方式是用C或者C++语言的功能直接用手写出代码，如果需要特别考虑时间和空间效率因素。由于这方面原因，当时强调的是自足的类而不是某种层次结构，强调将关键性操作在线化，强调这种类可以用在传统程序中，不需要重新设计也不需要重新训练程序员。特别地，这里都不企图给用户提供（通过在派生类里用覆盖虚函数的方式）修改这些类里的操作的手段。如果用户需要一个更一般的或者是修改过的类，那么可以通过将一个“标准”类作为构造块的方式，自己写出有关的类来。例如：

```
class String { // simple and efficient
    // ...
};

class My_string { // general and adaptable
    String rep;
    // ...
public:
    // ...
    virtual void append(const String&);
    virtual void append(const My_string&);
    // ...
};
```

8.3.1 I/O流库

C语言的printf函数族是高效的，一般说也是很方便的I/O机制。但无论如何它们不是类型安全的，也无法针对用户定义类型（类或枚举等）进行扩充。因此我那时就开始为printf函数族寻找一种类型安全、紧凑、可扩充的，而且是高效的替代物。部分地是受到来自Ada Rationale [IchBiah, 1979] 最后一页半的激励，那里论述说如果没有特殊语言特征的支持，你就无法得到一个简洁的类型安全的I/O库。我把这件事看作是一个挑战。结果就是这个I/O流库，它第一次是在1984年实现，发表在[Stroustrup, 1985]。那以后不久Dave Presotto又重新实现了这个流库，改进了它的性能。他的方式就是不再通过标准C函数，直接使用操作系统功能，而我原来的实现则是通过C标准库的函数。他在这样做的时候并没有改变流的界面。实际上，我就是在使用了一个上午或者更长时间之后才听说了Dave的修改。

为了解释I/O流，我们考虑下面的例子：

```
fprintf(stderr, "x = %s\n", x);
```

因为fprintf()实际上依赖于不经检查的参数，这些参数是根据格式串在运行时处理的，这样做当然不是类型安全的，还有[Stroustrup, 1985]

“当x具有像complex那样的用户定义类型时，就没有能‘使printf()理解’的方便方

式（例如%s和%d）去描述x的输出格式（14.2.1节）。典型情况是程序员需要另外定义独立的函数去打印复数，他可能写出下面这样的东西：

```
fprintf(stderr, "x = ");
put_complex(stderr, x);
fprintf(stderr, "\n");
```

这也很不好看。这种情况可能成为C++程序里的一种主要麻烦，因为那里要用到许多用户定义类型，以表示一个应用中所关注的/关键性的各种实体。

得到类型安全性和统一处理是可能的，可以通过对一组输出函数使用统一的重载函数名的方式。例如：

```
put(stderr, "x = ");
put(stderr, x);
put(stderr, "\n");
```

通过参数类型可以确定对一个具体参数到底该调用哪个‘put函数’。当然这种写法还是太啰嗦。C++的解决方案是使用一个输出流，将<<定义为它的输出运算符，写出来的样子如下：

```
cerr << "x = " << x << "\n";
```

其中的cerr是标准的错误输出流（等价于C里的stderr）。这样，如果x是具有值123的int变量，这个语句将打印出：

```
x = 123
```

后随的换行符号也送到标准错误输出流。

只要<<对x的类型有定义，那就可以用这种风格。用户也很容易为新类型定义<<运算符。所以，如果x具有用户定义的complex类型，值为(1,2.4)，那么上面的语句将向

```
x = (1,2.4)
```

cerr输出

在I/O流的实现中使用的完全是每个C++程序员都可以用的语言机制。与C语言一样，C++也没有把任何I/O能力构造到语言内部。I/O流功能是通过一个库提供的，其中并不包含任何超语言的魔力。”

提供一个输出运算符而不是一个命名的输出函数是Doug McIlroy的建议，所采用的形式类似于UNIX外壳的I/O重新定向运算符(>、>>、!等等)。这还要求有关运算符都返回它们的左运算对象，以便能在进一步的操作中使用：

“一个operator<<函数总返回作为它的调用参数的ostream的引用，这就使我们可以将另一个ostream应用^Θ到它上面。例如：

```
cerr << "x = " << x;
```

当x是个int时，它将被解释为：

```
(cerr.operator<<("x = ")).operator<<(x);
```

特别地，这意味着可以用一个输出语句打印出几个项来，它们将按照预想的顺序被打印：

^Θ 原书如此，看来是个错误。实际上是允许另一个operator<<应用，如下面例子所示。——译者注

从左到右。”

如果是采用一个常规的命名函数，用户将不得不写出类似前面最后一个例子那样的代码。有几个运算符曾被考虑作为输入和输出运算符：

“赋值运算符曾经是输入和输出运算符的一个候选对象，但是它约束的方式不对。也就是说，`cout=a=b`将被解释为`cout=(a=b)`。进一步说，大部分人似乎都喜欢用另外的输入和输出运算符。

也试验过运算符 `<` 和 `>`，但是“小于”和“大于”的意思在人们头脑里的印象太牢固了，新的I/O语句从实践的角度看很不容易读（对于 `<<` 和 `>>` 就不会出现这种情况）。除了这些情况外，在大部分键盘上符号 ‘`<`’ 正好在 ‘`,`’ 的上面，人们可能写出像下面这样的表达式：

```
cout < x , y , z;
```

对于这种东西不容易给出好的错误信息。”

实际上，现在我们可以用重载逗号运算符（11.5.5节）的方式来给出所需要的意义，但是按照1984年的C++定义还无法这样做，这会引起重复地写输出运算符。

在标准I/O流`cout`、`cin`等名字里的`c`是表示字符（character），它们都是为基于字符的I/O而设计的。

为了与Release 2.0相配合，Jerry Schwarz重新实现并部分地重新设计了这些流库，以便能更好地为一大类应用服务，也能更有效地进行文件的I/O [Schwarz, 1989]。其中一个重要的改进是采用了Andrew Koenig的操作符的思想 [Koenig, 1991] [Stroustrup, 1991]，用它们来控制格式，例如控制浮点数输出的精度或者整数输出的基数等。例如：

```
int i = 1234;
cout << i << ' '           // decimal by default: 1234
     << hex << i << ' '     // hexadecimal: 4d2
     << oct << i << '\n';   // octal: 2322
```

对这些流库的试验也是改变基本类型系统的一个重要原因，为此还修改了重载规则，使`char`值能够被作为字符来处理，而不是像C里那样作为一种小整数（11.2.1节）。例如：

```
char ch = 'b';
cout << 'a' << ch;
```

在Release 1.0里这将输出一串数字，它们所反映的是字符a和b对应的整数值。而在Release 2.0中则会像人们预期的那样输出ab。

与Cfront的Release 2.0一起发布的`iostream`库成为了后来其他提供商所发布的`iostream`的一种模式，也成为将要出来的标准`iostream`库的一部分（8.5节）。

8.3.2 并行支持

对并行的支持永远是各种库和语言扩充的丰富源泉。其中一个原因是许多空谈家都确信多处理器系统很快就会大大地发展起来。而按照我的判断，这种情况的广泛流行至少还得20年的时间。

多处理器系统正在变得越来越常见，但同时也出现了更令人吃惊的高速单处理器。这就蕴涵着至少两种形式的对并行性的需要：在单处理器上的多线程以及在多处理器上的多进程。此外，网络（包括WAN和LAN）也提出了它们的要求，还有大量存在的专用系统结构。正是由于这种多样性，我建议在C++里应该通过库的方式表述并行，而不是通过某种通用的语言特征。这种特征，比如说某种类似于Ada里作业的东西，对大部分用户而言都会是很不方便的。

设计出支持并行性的库，使之在使用方便性和效率上都能接近于内置的并行支持，这件事完全可能做到。基于这样的一些库，你当然可以支持各种各样的并行模型，用它们为需要多种不同模型的用户服务，肯定能比仅用一个内置的并行模型做得更好。我预计这将成为大部分人的选择方向，而由此引起的可移植性问题（社团中使用着多个不同的并行库）可以通过一个薄的界面类层次来处理。

有关并行支持库的例子可以参看 [Stroustrup, 1980b]、[Shopiro, 1987]、[Faust, 1990] 和[Parrington, 1990]。支持某种并行性形式的语言扩充的例子有Concurrent C++ [Gehani, 1988]、Compositional C++[Chandy, 1993] 和Micro C++[Buhr, 1992] 等。此外还存在大量专有的线程和轻载进程包。

1. 一个作业实例

作为通过库所提供的机制描述的并行程序的例子，我将演示一个用厄拉多塞筛法寻找素数的程序，其中为每个素数使用了一个作业。这个例子采用作业库 [Stroustrup, 1980b] 的队列携带整数，穿过一系列定义为作业的过滤器：

```
#include <task.h>
#include <iostream.h>
class Int_message : public object {
    int i;
public:
    Int_message(int n) : i(n) {}
    int val() { return i; }
};
```

这个作业系统的队列携带着由类object派生的类的信息，用到名字object说明这是一个很古老的库。在更新一点的程序里我会用模板来包裹这个队列，以提供类型安全性。但在这里我将保留早期作业库的使用风格。采用作业库的队列携带单个的整数当然是小题大做了，但这样做非常简单，队列能保证来自不同作业的put() 和get() 之间可以同步。这个队列的使用也显示了在模拟中或者在一个不能依赖共享存储的系统里如何将信息送来送去。

```
class sieve : public task {
    qtail* dest;
public:
    sieve(int prime, qhead* source);
};
```

由task派生的类可以与其他这样的作业并行执行。实际工作是在作业的建构函数和由它们调用的代码里完成的。在这个例子里，每个筛子就是一个作业，这种筛子从输入队列里取得一个数，并检查这个数能否被这个筛子所表示的素数整除。如果不行，这个筛子就把该数传递给下一个筛子。在没有下一个筛时，我们就又发现了一个素数，应该建立一个表示它的新筛子了：

```

sieve::sieve(int prime, qhead* source) : dest(0)
{
    cout << "prime\t" << prime << '\n';
    for(;;) {
        Int_message* p = (Int_message*) source->get();
        int n = p->val();
        if (n!=prime) {
            if (dest) {
                dest->put(p);
                continue;
            }
        }
        // prime found: make new sieve
        dest = new qtail;
        new sieve(n, dest->head());
    }
    delete p;
}
}

```

消息总是在自由空间里创建，最后由使用了这个消息的筛删除掉。这些作业在某种调度程序的控制之下运行。也就是说，这种作业系统不同于一个纯粹由协作程序构成的系统。因为在那种系统里，协作程序之间的所有控制转移都是显式进行的。

为了完成这个程序，我们需要用main()函数建立第一个筛：

```

int main()
{
    int n = 2;
    qtail* q = new qtail;
    new sieve(n, q->head()); // make first sieve
    for(;;) {
        q->put(new Int_message(++n));
        thistask->delay(1); // give sieves a chance to run
    }
}

```

这个程序将一直运行，直到耗尽了系统中的某种资源为止。我根本没在把它编写成能体面地死亡方面花功夫。这当然不是计算素数的有效方法，因为它对每个素数都要耗费一个作业，还要做许多次作业上下文的转换。这个程序可以作为一个模拟在单处理器系统上运行，所有作业共享统一的地址空间，也可以作为真正的并行程序而使用许多处理器。我测试它时是在一个DEC VAX上，按照模拟方式运行了一万个素数/作业。在C++里另一个更令人惊奇的厄拉多塞筛法的变形可以参看 [Sethi, 1989]。

2. 锁定机制

在处理并行性时，锁定的概念通常比作业的概念更基本。如果程序员能够说对某些数据需要采用互斥性的访问方式，那么通常就不需要知道实际上到底是哪个进程、作业、线程等。有些库得益于这种观点，它们都提供某种与锁定机制的标准接口。要将这个库移植到另一种新系统结构时，只需要正确实现这个接口，使之正确地与在那里遇到的并行性概念相配合。例如：

```

class Lock {
    // ...
}

```

```

public:
    Lock(Real_lock&); // grab lock
    ~Lock();           // release lock
};

void my_fct()
{
    Lock lck(q2lock); // grab lock associated with q2
    // use q2
}

```

在析构函数里释放锁定机制，这样就可以简化代码，并使它更加可靠。特别是这种风格能够与异常机制（16.5节）很好地相互作用。采用了这种风格的锁定机制，就能把关键性的数据结构或者策略做得完全不依赖于并行性的细节。

8.4 其他库

这里我将给出一个有关其他库的很短的表，目的是想说明C++库的多样性。还存在大量的库，每个月都可能出现几个新的C++库。看起来，某种形式的软件部件工业最终开始出现了——空谈家们早已许诺了许多年，或者是一直在哀叹其缺位。

下面提到的库都被归类为“其他的库”，只是因为它们没有对C++语言的开发产生重要影响，并不是对它们技术价值或者对用户的重要性的评价。实际上，一个库的建造人员为用户服务的最好方式通常就是细心而稳健地对待所使用的语言特征，这也是使库具有最大限度的可移植性的一种途径。

8.4.1 基础库

对于究竟什么构成了一个基础库，存在着两种几乎是相互正交的观点。有一种库被称为是水平的基础库，它们提供了一集基本的类，以便在每个应用中对每个程序员都能有所帮助。典型情况下这些类中包括各种基本的数据结构，如动态的和带检查的数组、表、关联数组、AVL树等等，还有许多常用的功能类，如字符串、正则表达式、日期和时间等。水平库的构造者通常都把很大功夫花在使库能够跨平台运行方面。

垂直的基础库则在做另一个方面的事情，目标是为某个特定的环境提供一个完整的服务集合，例如为X Window系统，或是为MS Windows，或为MacApp，或者是为一组这类环境。典型的垂直基础库也提供了可以在水平基础库里找到的那些类，但是它们的着重点在于那些能够利用所选环境的关键性特征的类。在这一端最重要的就是支持交互式用户界面和图形的有关类。与特定数据库系统的接口类也常常成为这种库的一个有机组成部分。通常这种垂直基础库的类之间互相融合，形成一个公共的框架，以至其中任何部分都很难取出来孤立使用。

我个人更赞同保持基础库中水平部分和垂直部分之间相互独立，以使其比较简单，并提供选择的可能。另一种考虑，既有技术因素也有商业因素，就是拼命地在那里做集成。

早期最重要的基础库是Keith Gorlen的NIH类库 [Gorlen, 1990]，它提供了类似Smalltalk的一集类；Mark Linton的Interview库 [Linton, 1987]，它使得在C++里可以方便地使用X Window系统。GNU C++ (G++) 出现时带有一个由Doug Lea设计的库，这个库在有效使用抽象基类机制方面表现出众 [Lea, 1993]。USL Standards Components [Carroll, 1993] 为数据结构和支持UNIX系统提供了一集高效率的具体类型，主要用在工业方面。Rogue Wave销售一个称为Tools++的库，这个库根源于Thomas Keffer和Bruce Eckel从1987年开始在华盛顿大学

开发的一集基础类 [Keffer, 1993]。Glockenspiel公司多年来一直为各种商业应用提供库 [Dearle, 1990]。Rational公司发售Booch Components的一个C++版本，这个库原来是由Grady Booch在Ada里设计和实现的。Grady Booch和Mike Vilot设计实现了其C++版本。那个Ada版本有125 000行不带注释的代码，与之成对照的是C++版本只有10 000行——继承机制与模板结合，能够成为组织库结构的极其强有力的机制，而又不损失性能和清晰性 [Booch, 1993]。

8.4.2 持续性和数据库

对于不同的人而言持续性意味着不同的东西。有些人不过是想有一个对象I/O包，就像许多库都提供的那样；一些人希望的是一种在文件和主存之间往返的无缝迁移机制；另一些人想要版本和交易的记录机制；还有一些人希望的是一个完整的分布式系统，带有完全的并发控制，以及为模式迁移提供的完善支持。正是由于这个原因，我认为对持续性的支持必须通过特殊的库，非标准的扩充，和/或第三方产品来完成。我看不到有任何对持续性机制进行标准化的希望，但是C++运行时的类型识别机制包含了若干“钩子”，对人们处理持续性问题应该有用处（14.2.5节）。

NIH库和GNU库都提供了基本的对象I/O机制。POET是C++商品的持续性库的一个例子。还有十来个面向对象的数据库是想提供给C++使用的，同时也是用C++实现的。ObjectStore, ONTOS [Cattel, 1991]，以及Versant都是这类东西的实例。

8.4.3 数值库

Rogue Wave [Kefer, 1992] 和Dyad提供了很大的一集类，其基本目标是为科学计算的用户使用。这种库的基本目的是使很复杂的数学能够以某个科学或工程领域的专家们感到方便而自然的形式来使用。下面是用到Sandia National Labs的RHALE++库的一个例子，这个库支持的是数学物理：

```
void Decompose(const double delt, SymTensor& V,
                Tensor& R, const Tensor& L)
{
    SymTensor D = Sym(L);
    AntiTensor W = Anti(L);
    Vector z = Dual(V*D);
    Vector omega = Dual(W) - 2.0*Inverse(V-Tr(V)*One)*z;
    AntiTensor Omega = 0.5*Dual(omega);

    R = Inverse(One-0.5*delt*Omega) * (One+0.5*delt*Omega)*R;
    V += delt*Sym(L*V-V*Omega);
}
```

按照 [Budge, 1992] 的说法，“这个代码是透明的，作为它的基础的类是多变的，而且很容易维护。一个熟悉极坐标分解算法的物理学家立即就能看出这个代码段的意思，根本不需要附加的文档。”

8.4.4 专用库

上面提出的库主要是为了支持某些更一般形式的程序设计。另一些专门支持特殊应用领

域的库对用户而言也具有同样的重要性。例如，人们可以找到公共域的、商业或公司的许多库，它们支持各种应用领域，例如流体动力学、分子生物学、通讯网络分析、电话操作控制台等等。对许多C++程序员而言，这种库也是C++证明其真正价值的地方，它们能简化程序设计，更少出现程序设计错误，减少维护的需要等等。最终用户大多根本就没有听说这种库，他们只是简单地从中获益。

下面是一个例子，这是对电路开关网络的一个模拟 [Eick, 1991]：

```
#include <simlib.h>

int trunks[] = { /* ... */ };
double load[] = { /* ... */ };
class LBA : public Policy { /* ... */ };

main()
{
    Sim sim;                                // event scheduler

    sim.network(new Network(trunks));        // create the network
    sim.traffic(new Traffic(load, 3.0));      // traffic matrix
    sim.policy(new LBA);                     // Lba routing policy

    sim.run(180);   // simulate 180 minutes

    cout<<sim;      // output results
}
```

这里涉及到的类或者就是SIMLIB的类，或者是用户由SIMLIB派生的类，它们定义网络、装载以及运行策略等，为某次特定的分析服务。

就像在前一节里物理学的例子一样，如果你是这个领域的专家，那么这里的代码本身就带有非常完美的意义表述。在这个情况中，有关领域很窄，因此这个库就是为一个高度专门化的应用领域里的人们服务的。

还有许多特殊的库，而它们实际上又很有普遍意义，例如支持图形学和可视化的库。但是本书并不想去列举所有的C++库，也不想给出一个完整的分类表。丰富多彩的C++库真让人应接不暇。

8.5 一个标准库

在说明了C++库的这种令人眼花缭乱的多样性之后，立即会浮现出一个问题：“哪些库将是标准的？”也就是说，在C++的标准里应该描述哪些库，并要求每个C++实现都必须提供呢？

首先，现在已经普遍使用的关键库都必须进行标准化。这意味着必须精确刻画C++与C语言标准库之间的准确接口，对iostream库也一样。此外，对基本语言的支持也必须给以精确的刻画，也就是说我们必须准确地刻画一些函数，如`::operator new(size_t)`和`set_new_handler()`，它们支持`new`运算符（10.6节）；`terminate()`和`unexpected()`支持异常处理（16.9节）；还有类`type_info`、`bad_cast`和`bad_typeid`，它们支持运行时的类型信息（14.2节）。

作为下一步，这个委员会还必须想想它是否有责任对公众关于“更多有用的标准库”的呼唤做出响应，例如string类等，而既不陷入委员会设计的泥沼，又不形成与C++的库工业的竞争。任何超出C语言库和iostream的库要想被这个委员会接受，它在性质上就必须是一个构件块，而不是一个更加雄心勃勃的应用框架。标准库的关键角色就是在那些分别开发的、更加雄心勃勃的库之间提供很容易使用的通讯。

按照这个思想，委员会已经接受了一个string库和一个宽字符的wstring库，还正在考虑将它们统一为一种允许任何东西的一般性的串模板。它还接受了一个数组类dynarray [Stal, 1993]，一个为建立固定大小的二进制位集合的模板类bits<N>，还有一个二进制位集合类bitstring，其大小可以改变。此外委员会还接受了一个复数类（我最初的complex类的后辈，见3.3节），并且在寻找向量类，其意向是支持数值/科学计算。由于这些标准类、它们的精确刻画、甚至它们的名字都还在轰轰烈烈的争论之中，我只好抑制住自己，不进一步给出细节和例子了。

我当然也想看到表和关联数组（映射）模板都出现在标准库里（9.2.3节），但是，就像在Release 1.0一样，这些类也可能因为时间关系，赶不上完成核心语言这个正在迫近的事件了。^Θ

Θ 在这里，我带着激动和喜悦来收回我的话。委员会最终抓住了时机，通过了由Alex Stepanov设计的包含各种容器、遍历器和算法的异常出色的库。这个库常被称做STL，它是一个优雅、高效、语义合理、经过良好测试的有关各种容器及其使用的框架（Alex Stepanov和Meng Lee, *The Standard Template Library*（标准模板库），HP实验室技术报告HPL-94-34(R.1)，1994年8月。Mike Vilot, *The C++ Report*, 1994年10月）。当然，STL也包含了映射和表类，并且包容了上面提到的dynarray、bits、bitstring类。此外，委员会还通过了一个支持数值/科学计算用的向量类，这个类是基于Sandia Labs的Ken Budge的一个建议书。

第9章 展望

你无法两次游过同一条河。

——赫拉克利特

C++在其预期领域取得了成功吗？——C++是不是一个统一的语言——什么东西本不该是现在的样子？——什么东西本应该加进来？——什么是最大失误？——C++仅仅是一座桥梁吗？——C++对什么最合适？——什么能使C++更有效？

9.1 引言

与我最喜欢做的事情不同，这一章更具推测性，更多地依赖于个人观点，也更具一般性：我实际上更喜欢描述那些已经完成的工作和试验。但无论如何，在这章里要回答一些常见的问题，并陈述一些论点，这些东西在讨论C++时总是不断地出现。本章由三个相关的部分组成：

- 一个回顾。试着去评价今天的C++在哪些地方与它原来的目标有关系，或者与它原本可能成为的东西有关系（9.2节）。
- 对软件开发和程序设计语言领域里未来可能出现的问题的一个考察，看看C++可能如何面对这些问题，并使自己能适应于这个不断改变的世界（9.3节）。
- 对一些领域的考察。在这些领域里C++及其使用方式可能做较大改进，以使C++成为一个更好的工具（9.4节）。

讨论未来发展总是很冒险的，但这也是一个值得去冒的险：语言设计的一部分就是必须预先考虑未来的问题。

9.2 回顾

人们常说，事后的认识是最精确的科学。这个断言实际上基于一些错误假设，即认为我们已经知道了关于过去的所有相关事实，也知道了事情的当前状态，而且还掌握着一个合适的超然的观察点，能够从这个位置出发去做出判断。典型情况是所有这些条件都根本不成立。这样，要对一个大规模使用的程序设计语言这样大型的、复杂的、而且是动态变化的事物做一个回顾，不可能只是一些关于事实的陈述。但无论如何让我试着转过身，来回答一些很困难的问题：

- 1) C++在其预期领域取得了成功吗？
- 2) C++是不是一个统一的语言？
- 3) 什么是其中最大的失误？

自然，对这三个问题的回答都只能是相对的。我的基本回答是：“是”，“是”，“没有在Release 1.0的同时发布一个大的库”。

9.2.1 C++在其预期领域取得了成功吗

“C++是一种通用的程序设计语言，其设计就是为了使认真的程序员能觉得编程变得更愉快。”[Stroustrup, 1986b]。按照这个目标，C++确实是成功了。特别是，它的成功在于使那些具有合理的教育和经验的程序员能在更高的抽象层次上写程序（“就像在Simula里那样”），而又不损失能与C语言媲美的效率。特别是，对那些同时受到许多因素支配的应用，如时间、空间、内在复杂性、以及来自执行环境的约束等等。

更一般地说，C++使得面向对象的程序设计和数据抽象等等真正地用到了实际开发者的社团里。而在那个时候以前，这些技术和支持它们的语言（如Smalltalk、Clu、Simula、Ada、OO Lisp方言等）还受到蔑视，甚至被嘲笑为“根本不适合实际问题的昂贵玩具”。C++做了三件事去跨过这些难以对付的障碍：

- C++能产生运行时间和空间特性极好的代码，这种代码能与在本领域中广泛认可的领先者C语言比赛。任何东西要能与C比赛或者超过它都必须是足够快的。任何不能与之竞争的东西都可能并且也将——如果不考虑必要性或仅由于偏见——被弃之一旁。C++不但能从按传统方式组织的代码中产生这种性能，对基于数据抽象和面向对象技术的代码也一样。
- C++使人可以将这种代码集成到常规系统里，并能在传统系统上生成。具有常规的可移植性也是最根本的问题。C++有能力与现存的代码共生，与传统的工具，例如排错系统和编辑系统共生。
- C++允许人们逐步转移到新的程序设计技术上来。学习新技术需要时间，公司根本就承受不起让一大批程序员在学习期间完全没有产出的代价。它们也不能承受由于程序员过分热情地而又是错误地应用了一些还没有完全把握的新思想，从而导致项目失败的代价。

C++使面向对象的程序设计和数据抽象变便宜了，可以使用了。

在这个成功中，C++不仅帮助了它自己的用户社团。它还为所有支持不同形式的面向对象的程序设计和数据抽象的语言提供了巨大的推动力。C++当然不是对所有人的万能灵药，也从来没有承诺过一些人针对别的语言做过的承诺或者其他东西。它从来就不想这样，我也没做过过分的承诺。但是无论如何，C++确实实现了自己的承诺，它冲破了挡在所有使程序员能在更高的抽象层次上工作的语言面前的一堵不信任之墙。通过做了这件事，C++为它自己，也为另一些语言——其支持者只是把C++看成是竞争对手——打开了许多扇门。还有，C++也帮助了其他语言的用户，因为它给那些语言的实现者们一种强烈的刺激，推动他们去改进其性能和灵活性。

9.2.2 C++是不是一个统一的语言

简而言之，我对这个语言很满意，但是同意我的人并不很多。确实存在着许多细节，如果可能的话我也愿意去改进。但无论如何，一个基于带虚函数的类的静态类型语言，提供了为低级程序设计所需要的功能，这个概念是合理的。此外还有，这里的主要特征都能在相互支持的方式下一起工作。

1. 什么东西本不该而且也可以不是现在的样子

什么样的东西可能是一个比C++更好的语言，又能够对付C++要处理的那些问题？考虑第

一级的决策 (1.1节, 2.3节, 2.7节):

- 使用静态类型检查和类似Simula的类。
- 语言和环境之间清晰的隔离。
- C源代码兼容性 (“尽可能靠近C语言”)。
- 在与C连接和布局上的兼容性 (“真正的局部变量”)。
- 不依赖废料收集。

我始终把静态类型检查看作是最根本的东西, 无论是对于好的设计, 还是对于运行时的效率。如果我要再去为C++今天所做的事情设计一个新语言, 我还是要追随Simula的类型检查和继承模型, 而不会去采用Smalltalk或者Lisp的模型。正如我多次说过的, “如果我想去模仿Smalltalk, 一定是希望能构造出一个更好模仿物。但是Smalltalk本身已经是最好的Smalltalk了。如果你想用Smalltalk, 那么就去用它” [Stroustrup, 1990]。同时有静态类型检查和动态类型识别 (例如, 在虚函数调用中), 与只有静态或者只有动态类型检查的语言相比, 这样做实际上意味着有一些很困难的权衡与选择问题。静态和动态类型模型不可能完全相同, 这样做必然会带来某些复杂性或者不优雅, 只支持一个类型模型就可能避免这些东西。但无论如何我并不想写只采用一种模型的程序。

我也一直把环境和语言的隔离看成是最根本的东西。我并不希望只使用一个语言, 一集工具, 或者一个操作系统。为了能提供选择, 隔离就是最基本的东西。而一旦有了这种隔离之后, 人们就可以提供不同的环境去适应不同的工作和不同的需要, 无论是技术支持方面、资源消耗方面, 还是可移植性方面。

我们的历史从来不是一张白纸。仅仅提供一些新东西是不够的, 我们还需要使人能够从旧工具和旧思想转变到新的方面去。这样, 如果C不摆在那里, C++不需要与之兼容, 那么我也一定会选择去与另外的某种语言大体上兼容。当然, 任何兼容性都隐含着某些很丑陋的东西。由于构筑在C语言之上, C++继承到了某些语法怪癖、有关内部类型转换的一些相当混乱的规则等等, 这些不完美的东西还将继续引起争论。但是其他可能的选择——例如作为一个基于C的语言却又带有重大的不兼容性, 或者是想让一个完全从空白中建造起来的语言得到广泛的应用——将会遇到更多得多的麻烦。特别是, 与C在连接和库方面的兼容性是极端重要的。因为能有与C语言的连接兼容性, 也就意味着C++能够与大部分其他语言连接, 只要那些语言提供了与C代码的联接机制。

一个语言到底是应该对变量采用引用语义 (也就是说, 变量实际上就是一些指向放置在其他地方的对象的指针), 就像Smalltalk和Modula-3的情况; 或者是允许真正的局部变量, 像C和Pascal里这样。这个问题也很关键, 它关系到与其他语言共存、运行效率、存储管理以及多态类型使用等方面的诸多问题。Simula避开了这个问题, 它 (只) 对类对象采用引用, 而 (只) 对内部类型的对象采用真正的局部变量。我认为, 对于一个语言的设计能否同时提供引用和局部变量的优点, 而又不太丑陋, 这个问题并没有一个定论。如果再给我一次选择, 一边是优雅, 而另一边是同时具有引用和真正局部变量的优点, 我还会选择采用两种变量。

一个语言应该直接支持废料收集吗, 例如像Modula-3那样? 如果是, 那么C++能不能在提供了废料收集的同时又能达到它的目标呢? 废料收集是极好的东西, 如果你能负担得起使用它的额外代价。这也就是说, 如果能有废料收集当然是很好的, 但是从运行时间、实时以

及移植的方面看废料收集的代价太高了（这个代价到底怎样，这又是个很混乱的有争议的题目）。由于这些，每时每刻都必须为废料收集付出代价就不是一件幸事了。C++允许废料收集作为可选的机制 [2nd, 第466~468页]。有几个带有废料收集的C++实现正在进行之中。我期望在几年内，我的一些C++程序可以依靠废料收集（10.7节），但不是所有的程序。但是，我也始终相信（在过去许多年里曾多次重新考虑这个问题之后），如果C++的初始设计就依赖于废料收集的话，它也早就胎死腹中了。

2. 什么东西本应该排除在外

甚至在 [Stroustrup, 1980] 里就提出了有关带类的C可能变得太大的担心。我认为，在对C++的所有意愿表，“一个较小的语言”必须排在第一。然而人们向我和委员会提出的扩充建议书却像洪水一样。我看不出能够除掉C++的某个主要部分，而又不使得某些重要的技术变得缺乏支持。甚至在完全不考虑兼容性的问题的情况下，我们也只能对C++的基本机制做很少的化简，这些当然是在C++的C子集中——有时我们甚至忘记了C语言本身也是个相当大而且非常复杂的语言。

C++规模很大的基本原因在于它要支持以不止一种方式、不止一种程序设计范型去写程序。从某种观点看，C++实际上是将三个语言合为一体：

- 一个类似C的语言（支持低级程序设计）
- 一个类似Ada的语言（支持抽象数据类型程序设计）
- 一个类似Simula的语言（支持面向对象的程序设计）
- 将上述特征综合成一个有机整体所需要的东西

也可以在一个类似C的语言里按照这些风格中的任何一个来写程序，但是C并没有对数据抽象或面向对象程序设计的直接支持。而在另一方面，C++则是直接地支持所有这些不同的方式。

始终存在着多种设计选择。但是在大多数语言里，语言的设计者都已经为你做了选择。对C++我就没有这么做，而是把选择的权利交给了你。对那些相信只有惟一一种做事情的正确风格的人而言，这种灵活性自然是很讨厌的。这样做也会吓走了一些初学者和教师，他们可能觉得一个好语言就是那种在一个星期里就能完全理解的东西。C++不是这样的一种语言，它的设计就是为了给专业人员提供一个工具箱。抱怨在这里的特征太多了，就像是一个“门外汉”在窥视了一个室内装饰工的工具箱之后，抱怨说根本不可能需要这么多种小锤子一样。

每种不只有简单应用的语言都需要成长，以满足其用户社团的需要。这就必然意味着复杂性的不断增长。C++正是这种趋势的一部分，它趋向于一个具有更大复杂性的语言，以便能处理人们想解决的更加复杂得多的程序任务。如果复杂性不出现在语言本身，那么它必定出现在库或者工具方面。许多语言/系统（与它们初始的简单情况相比较）都有了巨大的增长，如Ada、Eiffel、Lisp（CLOS）以及Smalltalk等。由于C++强调的是静态类型检查，复杂性增长中的许多情况都是以语言扩充的形式出现的。

C++是为严肃认真的程序员设计的，它也将为能帮助他们面对更大更复杂的工作而发展成长。这样发展的结果确实很可能使新来的人们不知所措，甚至是对一些经验丰富的新人。我一直在试着尽可能地减小C++语言规模的实际影响，使人们可能分阶段的学习和使用C++（7.2节）。也通过避免“分布式的增肥”使大语言对性能的常见的负面影响达到最小。

(4.5节)。

3. 什么东西本应该加进来

如前, 基本原则是越少越好。有一封以C++标准化委员会的扩充工作组名义发表的信针对这个问题说 [Stroustrup, 1992b]:

“首先, 我们想试着劝阻你不要对C++语言提出扩充建议。按照我们的评价, C++已经是太大太复杂了, 有成百万行的代码已经放在那里, 我们需要努力不去打破它们。对语言的所有修改都必须经过极其慎重的考虑, 给它增加东西更是令人胆战心惊。只要可能, 我们将更愿意看到以程序设计技术或者库函数代替语言的扩充。”

许多程序员社团都希望看到他们最喜爱的语言特征或者库类能够传播进C++里。不幸的是, 如果为各种各样的社团添加有用的特征, 就会使C++变成一个没有内在统一性的特征集合。C++本来就不完美, 而增加特征很容易把它弄得更糟而不是更好。”

那么按照这种说法, 哪些特征因为没有出场已经给我们造成了麻烦呢? 而哪些又正在争论之中, 可能在今后几年里进入C++? 简单地说, 在这本书里描述的特征(包括在第二部分里描述的那些, 如模板、异常、名字空间和运行时类型识别)对我来说已经足够了。我还希望有可以选择的废料收集, 但把它归类到实现质量问题, 而不是语言特征。

9.2.3 什么是最失误

在我的心里, 能够竞争最大失误地位的只有一件事情。Release 1.0和我的书的第一版 [Stroustrup, 1986] 那时都应该推迟一些, 直到有了一个比较大的库, 包含一些基础类, 如单向和双向链表、一个关联数组类、一个检查范围的数组类, 还可以包括一个简单的字符串类。缺少了这些, 就导致每个人都需要重新发明轮子, 也导致在这些最基础的类方面不必要的多样性。它也导致精力的严重分散。为了企图自己去构筑这些基础类, 太多的新程序员都在还没有掌握C++最基本的东西之前, 就开始去尝试那些为构造出好的基础类而需要的“高级”特征。还有, 在由于没有模板的支持而造成的库的内在缺陷有关的技术和工具方面花掉了太多的时间。

我能避免这些问题吗? 从某种意义上说, 很明显, 我应该能。我的书原来计划包括三个有关库的章, 一章讨论流库, 另一章讨论容器库, 还有一章讨论作业库。我大致上知道我想要的是什么。不幸的是那时我太疲惫了, 没有某种形式的模板无法做出容器类。也没有想出通过预处理器或者一个不完全编译器去“伪造”模板的想法。

9.3 仅仅是一座桥梁吗

我构造C++是想作为一座桥梁, 程序员能够借助于它, 从传统的程序设计过渡到依赖于数据抽象和面向对象的程序设计。C++在此之外还能有它自己的未来吗? C++仅仅是一座桥梁吗? 一旦跨到另一个世界里, 在那里数据抽象和面向对象的程序设计并不是那么自然, C++所提供的那些特征还有其本身的价值吗? 此外, 它由C语言继承来的那些东西会不会变成一种致命的义务? 还有, 假定对上面问题得到的是正面回答, 那么在今后的十年里, 我们为那些并不关心C兼容性的用户所做的任何事情, 都不会对那些始终关心这个问题的人们造成损害吗?

语言的存在就是为了帮助人们解决问题。如果一个语言开始很成功，它就会生存下去，只要人们继续面临着这个语言能够帮助他们解决的同一类问题。进一步说，只要没有其他语言能够在同类问题上提供明显优于它的解，它就应该还能繁荣兴旺。这样，问题就变成：

- C++帮助我们解决的问题仍然是实在的吗？
- 明显优于它的解出现了吗？
- C++将为新的问题提供更好的解吗？

我的简单回答是“许多还将是”、“慢慢地”和“是的”。

9.3.1 在一个很长的时期里我们还需要这座桥梁

为了能在面向对象的编程和面向对象的设计等方面达到我所设想的那样精通和成熟的程度，人们还需要用很长的时间。向C++的迁移在由现在开始的五年里还不可能完成。C++作为一座桥梁和作为一种混合设计开发媒介的角色至少在20世纪里还会延续，它作为一种维护老代码和对它们升级的媒介作用将延续更长得多的时间。

应该清醒地认识到，在许多地方从汇编语言到C语言的转变还没有完成。按照同样的方式，从C到C++的转换可能要持续更长时间。当然这里也正是C++的长处之所在。对那些需要某种纯粹C语言风格的人们，这些风格也都已经能在C++里使用，而且同样有效。支持这些风格（无论是在转变过程中，还是在那些它们最合适的地方）正是C++最基本目标的一部分。

9.3.2 如果C++是答案，那么问题是什么

根本不存在这个问题。C++是一种通用程序设计语言——或者至少说是一种多用途语言。这就意味着对于任一个特定的问题，你总可以构造出一个语言或者一个系统，使它成为比C++更好的一条解决途径。C++的长处，更多地在于它对许多问题都是很好的解决途径，而不在于对某个特定问题是最好的解决途径。例如，与C语言类似，C++对于低级的系统也是一个绝好的语言，通常在这类工作中性能超过其他任何高级语言。当然，对于多数机器系统结构，一个好的汇编程序员总能生产出比很好的C++编译系统还要小许多、快许多的代码。通常这并不重要，因为在一个很复杂的系统里，这种存在显著差异的部分所占的比例很小，而如果整个系统都用汇编语言写出来，那将是无法负担的，也是无法维护的。

我发现，要设想出一个应用领域，在那里人们不可能构造出某种优于C++的特殊语言，同时也优于任何通用的程序设计语言，这是极其困难的事情。这样，大部分通用程序设计语言最希望做的也就是成为“每个人的第二选择”。

说了这些之后，下面我要考察一些领域，在这些领域里C++有其根本性的优势：

- 低级系统程序设计
- 高级系统程序设计
- 嵌入式代码
- 数值/科学计算
- 一般应用程序设计

这些类别并不是互相分离的，对它们也不存在已经被广泛接受的定义。C++将继续是所有这些领域中的一个很好选择。进一步说，任何语言要想成为一个好选择，那么在所提供的基本服务的层次上，它看起来会很像C++——当然，或许不是在语法或者语义细节的层次上。上面

列举的这些领域并没有穷尽已经成功地使了C++的那些应用类，但是它们也说明了一些关键性问题，C++要继续繁荣发展就必须面对它们。

1. 低级系统程序设计

C++是目前能用的最好的低级程序设计语言，它结合了C语言在这个领域中的优点，还有在不付出额外的运行时间空间代价下完成简单的数据抽象，以及驾驭具有这类特点的大型程序的能力。还没有任何新语言在这个领域里正在显示出远远超过C++，因而有可能取代它。涉及到低层次部件的系统程序设计将仍然是C++的优势领域，在这个领域里，C++充当的是作为更好的C的角色。今后一些年里，C++在这个领域中真正的竞争对手将始终是C语言，在这里C++将是更好的选择，因为它是更好的C。我预计低级系统程序设计将缓慢地——只是缓慢地——降低其重要性，而且将一直是C++的重要优势领域。由于这个原因，我们也必须小心，不能把C++语言或者系统“改进”到仅仅是一个高级语言的程度。

2. 高级系统程序设计

传统系统程序的规模和复杂性一直在迅速增长。这方面的例子如操作系统核心、网络管理系统、编译系统、电子邮件系统、文字排版系统、图像和声音的编排操作系统、通讯系统、用户界面、数据库系统等等。随之，传统上对底层效率的考虑也逐渐地让位于对系统整体结构的关心。效率当然还是重要的，但已经变成了第二位的东西。除非更大的系统能够很经济地构造出来并能很好地维护，否则效率就毫无用处。

C++提供的数据抽象和面向对象程序设计的功能正是直面了这种关心。对于从事这类应用的程序员而言，模板、名字空间和异常处理将变得越来越重要。将底层函数、子系统和库里不得不违反类型规则的东西隔离，这一点也将变得更加关键，因为这种技术能保证应用中的主要代码是类型安全的，因此也成为易维护的。我预计高层系统程序设计在今后许多年里仍将非常重要，它也将成为C++的一个优势领域。

许多其他语言也能够很好地为高层系统程序设计服务，这方面的例子如Ada9X、Eiffel、Modula-3。除了支持废料收集和并行性之外，这些语言在所提供的基本机制方面大致与C++等价。自然，有关个别语言特征以及它们集成到语言里的质量都可以进一步讨论，大部分程序员也将会有强烈的偏好。但不管怎样，只要存在质量充分好的实现，这些语言中的每一个都能支持范围广泛的系统应用。制约着发展的实际问题与程序设计语言的技术细节无关，而在于那些例如管理、设计技术以及程序员的教育。C++也倾向于有一些优势，例如在运行时间效率、灵活性、可用性及用户社团，这些都使它站在竞争的前列。

对某些更大的系统应用，废料收集是一个重要的优点；而对另一些它则是障碍。除非C++实现提供了可选择的废料收集，否则它就会在某些领域里受到这种缺点的伤害。我确信支持可选废料收集的实现将会变得很常见。

3. 嵌入式系统

系统程序设计的另一个领域值得专门提出来，那就是嵌入式代码，也就是说，运行在计算机化的设备里面的那些程序。例如照相机、汽车、火箭、电话交换机等。我推测这类工作将变得越来越重要，这种系统将是低层和高层系统程序设计的混合，C++对这种东西是最合适的。不同应用和不同组织将提出许多不同方面的要求，而专用的语言将很难满足所有这些要求。有些设计将紧紧地依赖于异常机制；而另一些可能会禁止异常，因为它们根本无法预计。与此类似，对于存储管理的需求也可能从“不允许动态存储”直到“必须使用自动废料

收集”。此外还可能使用各种各样的并行模型。C++是语言而不是一个系统的性质在这里就非常重要了，这就使它有可能去适应特殊的系统，为特殊的执行环境生成代码。对于有些项目，可以在独立的程序开发环境里或者在商品硬件的模拟器上运行C++也是至关重要的。C++程序可以放进ROM里的事实在过去就已经非常重要了。对于C++在为各种各样计算机化的物件编程的这个领域，我一直有很高的期望。在这个领域里C++还是可以站在C语言传统实力的肩膀之上。

4. 数值/科学计算

单从程序员的人数看，数值/科学计算是个相对较小的领域，但它又是非常有趣非常重要的。我已经看到了一种向高级算法移动的趋势，这也就更有利于能够表示多样性的数据结构并能高效地使用它们的语言。这种逐步增长的对于灵活性的需求将成为一种力量，能够抵消Fortran在基本向量运算上的一些优势。更重要的是，如果需要的话，或者就是为了方便，C++程序可以直接调用Fortran或汇编语言写的例行程序。希望把数值程序集成到很大的应用中，又创造出一种适合C++的需求。例如，对于强调非数值应用方面的需要，如可视化、模拟、数据库访问或者实时数据获取等等，Fortran在低层次计算方面的优点可能就会变得不足道了。

5. 通用应用程序设计

如果在某些应用里没有重要的系统程序设计部件，或者对运行时间空间的要求都不太重要，那么用C++就不一定很理想了。无论如何，在库和可能的废料收集的支持下，C++也常常能成为一个可行的工具。

我预计，对于许多这样的应用领域，专用的语言、程序生成器和直接操作的工具将可能占据主导地位。例如，如果你通过菜单组合出屏幕布局的一个实际例子之后，某个程序就能帮助你生成出用户界面的代码，那么你为什么还要自己写程序去产生它呢？类似地，当你可以用一个高层次的专用语言做高等数学时，为什么还要写Fortran或者C++去做它们呢？当然，即使在这些情况中，这些高级的语言、工具、或者生成器也需要用某种适当的语言来实现，它们也不时需要生成某种较低级语言的代码去实际地执行有关动作。C++常常又能适合对实现语言和目标语言的各种需求，所以我预料C++将扮演的一个重要角色就是作为高级语言和工具的实现语言。这些是C++从C语言那里继承来的另一批角色。C++语言的一些细节，例如允许在几乎任何地方声明变量，再与某些主要的程序组织特性（如名字空间）相结合，将使它比C语言更适合作为一种目标语言。

高级的语言和工具倾向于专门化。因此，好的这类东西都会提供一些功能，允许用户扩充或者修改系统的默认行为，所采用的方式是增加一些用某种低级语言写的代码。C++的抽象机制可以用于将C++代码平滑地连接到高级工具所提供的框架中去。

6. 混合系统

C++最显著的强项，就在于它能够在组合了多种不同应用的各个方面的系统或者组织里很好地工作。我推测，最重要的系统或者组织都需要这种类型的组合。用户界面通常都需要图形；特殊的应用常常依赖于特定的语言或程序生成器；模拟器和分析子系统需要计算；通讯子系统需要大量的系统程序设计；大部分大系统都需要数据库；特殊硬件要求低级操作。在所有这些领域——以及其他地方——C++至少能成为第二种选择。综合起来看，如果考虑的是一个主要语言，C++就会成为第一选择。

所有的语言都要死亡，或者将为迎接新的挑战而改变。一个有着极大的而又活跃的用户社团的语言将总是去改变而不是死亡。这也就是发生在C语言上的事情，由此产生了C++。而到某一天，这件事也可能发生在C++身上。C++是一种相对年轻的语言，尽管如此，考虑它从前驱者那里得来的并需要由后来者补偿的实力和弱点也是很有价值的。

C++并不完美；它没有想设计成完美的东西，任何其他通用语言也不可能。但无论如何C++已经足够地好了，因此不会被另一个类似的语言所取代。只有某个从根本上不同的语言才可能提供显著而充分的优点，使其成为一个明显的更强者。仅仅是作为一个更好的C++还不足以导致一个大转变，这就是为什么C++不能仅仅是一个更好的C的原因：如果C++没有提供重要的新的写程序方式，程序员就根本不值得从C的方面转过来。这也是为什么Pascal和Modula-2无法成为C的另一种选择的原因，虽然学术社会中很重要的一部分许多年来一直在推行这些语言：因为它们与C并没有重要差别，其优点并不那么显著。还有，如果某个更好但又没有截然差别的东西出现，一个热爱着原有东西的多样化社会将简单地吸收那些新的思想和特征。在C++的初始设计以及它发展到当前这个语言的演化过程中，就存在许多这方面的例子。

我还看不到在很近的未来有一种从根本上不同的语言能够在C++所覆盖的领域上取代它——只有一些以不同的方式提供了基本上类似的特征集合的语言，壁龛语言，以及试验性语言。我预计，这些试验性语言中的某一些将随着时间而成长，提供一些重要的改进，超过今天的C++和经过今后一些年的演化之后的C++语言。

9.4 什么能够使C++更有效

在软件开发的世界里，根本就没有骄傲自满的位置。在这些年里，人们期望的增长总是大大地超过硬件和软件的难以令人置信的增长。我看不到有任何理由说这种情况会很快改变。要使C++的实现更有助于它的用户，还有许多事情要做。程序员和设计师也有许多东西要学，以便使自己的工作更加有效。在这里，我将冒险给出一些见解，谈谈我认为要使C++程序设计更有效应该做些什么。

9.4.1 稳定性和标准

语言定义，关键性的库和界面的稳定性应该列在未来进步的需求表上最高的位置。ANSI/ISO C++标准将提供前者，许多组织和公司在后一方面工作，在各种领域中，如操作系统界面、动态连接库、数据库界面等等。我等待着某一天——在未来不远——这本书里描述的C++语言将成为在各种重要平台上都可以使用的东西，这会大大地推动库和工具工业的发展。

人们当然还会继续要求新的特征，但我已经可以在这里所描述的C++中生活了。我想写出大部分产品代码的程序员也应该可以。特别值得提醒一下，没有任何个别的特征对于生产好代码而言是最基本的东西——无论你怎样给出“好”的定义。

9.4.2 教育和技术

对于C++及其所有的应用领域而言，我认为对进步最有潜力的事情就是学习新的设计技术

和编程技术。从原则上说，更有效地使用C++是最容易获得的进步，也最廉价，昂贵的工具并不是必需的。在另一方面，改变思维习惯也不是很容易的事情。对大部分程序员而言，所需要的并不是简单的有关新语法的训练，而是有关新概念的教育。看一看7.2节，读一本讨论了设计问题的教科书，例如[2nd]或者[Booch, 1993]。我预计在今后几年里将看到在设计和编程技术方面的重大进步，这里当然没有拖延的理由。我们中的大多数已经在一个或者几个领域里大大地落后于现状了，从一些阅读和当前的试验中，我们可以得到重要的收获，此战斗在标准和工具的血刃上更加乐趣无穷。

9.4.3 系统方面的问题

C++是语言而不是一个系统，在许多环境里这都是一个强项，通过所提供的各种工具，可以组成一个完整的开发和运行环境。但无论如何，语言和环境之间的界面很难穿过这种分划之间的孔隙，这已经导致在某些领域中的发展进程非常缓慢，令人失望，例如增量编译、动态装入等。大体上说，人们还没有做什么事情，只是依赖为C语言设计的那些机制，或者只是在出发点极端宽泛，足以支持“所有面向对象的程序设计语言”的机制上做工作。从C++程序员的观点看，这样做取得的成果是相当可怜的。

将C++与动态连接集成起来的早期试验已经说明这是很有希望的，因此我曾经预计类的动态连接在几年之前就能够普及。例如我们在1990年就有了一种能运行的技术，完成基于抽象类的高效而类型安全的增量式连接[Stroustrup, 1987d][Dorward, 1990]。这种技术在实际系统里使用得不多。而抽象类在维护防火墙、减少在修改之后的重新编译等等方面正在变得越来越重要。一般地说，它能够使组合式地同时使用多个来源的软件部件变得更容易些(13.2.2节)。

还有一个使人烦恼的重要问题是支持软件的演化，因为无法很好地适应于现实情况，程序员世界实际上分化为一些互不相干的关心领域。从根本上说，问题是，一旦某个库已经在使用中，你要想修改这个库的实现，必要的条件就是库的用户并不依赖于库的实现细节，或者他们愿意并且也能够按照库的新版本重新编译自己的代码。各种对象模型，例如Microsoft的OLE2，IBM的SOM，Object Management Group的CORBA，其目标都是针对这个问题，其方式都是提供一个隐藏起实现细节的界面，并假定这个界面是与语言无关的。语言无关性给C++的程序员强加了许多麻烦，在通常情况还会增加时间和空间开销。此外，软件工业的每个重要分支看起来都自己的对付这个问题的“标准”。只有时间能告诉我们，这些技术将在什么程度上帮助或者是阻挠C++的程序员们。名字空间机制提供了一种在C++自身内部解决界面演化问题的途径(17.4.4节)。

我已经不太情愿地考虑接受这个认识：有些与系统有关的事项可能在C++里面处理起来更好一些。与系统有关的事项，例如类的动态连接、界面演化等等，从逻辑上说都不属于语言，从技术基础上看，基于语言的解决方案并不是更可取的。但无论如何，语言提供了惟一的公共论坛，在这里真正标准的解决方案有可能被接受。例如，Fortran和C的调用界面已经成为语言间相互调用的一种实际标准。之所以如此，是因为C和Fortran的使用广泛，也因为它们的调用界面简单而高效——是最小公因子。我并不喜欢这个结论，因为这里隐含着关于在系统里使用多种语言的一个障碍，要求一个语言所支持的机制必须变成其他语言都接受的一种标准。

9.4.4 在文件和语法之外

让我勾画一下自己喜欢看到的针对C++的程序设计环境。首先我希望有增量编译。当我做了一点小修改后，我希望“系统”能够注意到有关的修改是很小的，可以在一秒之内就完成新版本的编译，使之能够去运行。与此类似，我希望能做简单的询问，例如“请给我显示出这个f的声明？”“在这个作用域里的f是什么？”“这个+的使用将解析成什么？”“从类Shape派生的类有哪些？”以及“在这个块的最后有哪些析构函数将被调用？”还希望立刻就能得到回答。

一个C++程序里包含着大量的信息，在典型的环境中，这些信息只是由编译程序使用。我希望这些信息也能够在程序员的掌握之中。当然，大部分程序员还只是把一个C++程序看成一集源文件或者一个字符串。这实际上是混淆了一种表示和由它所表示的东西。一个程序是许多类型、函数、语句等等的一个汇集。为了能适应传统的程序设计环境，这些概念都用文件中的字符串表示。

将C++的实现放在基于字符的工具之上，这已经成为发展的一个主要障碍。如果你在对某个函数做了点小修改之后，必须预处理并重新编译这个函数所在文件直接或者间接包含进的所有头文件，那么就不可能有一秒钟的重新编译。一些避免冗余编译的技术已经开始出现了。按照我的看法，免除传统的源程序正文，基于一种抽象的内部表示去构造工具是最有希望和最有趣的途径。在 [Murray, 1992][Koenig, 1992] 里可以看到这种表示的一个早期版本。当然，我们还是需要用正文作为输入和供人阅读，但这种正文是很容易吸人系统的，在需要时也很容易重构。它不应该成为最基本的东西。根据C++语法依照某种缩行偏好格式化的正文不过是观察一个程序的许多可能方式之一。这个概念的一个最简单的应用就是允许你按照你喜欢的编排风格去看一个程序，而我同时又可以按照我的喜好去看同一个程序。

这种非正文表示的一个重要应用将成为高级语言的代码生成器、程序生成器、或者直接操作工具等的目标。它将使这些工具能够越过传统的C++语法，甚至能成为一种工具，使C++语言远远避开其语法的一些扭曲得特别厉害的方面。我在维持C++的类型系统及其语义的清晰性方面事情远胜于对C++的语法所做的事情。在C++里面存在着一个更小一些和更清晰一些的语言，它正在挣扎着浮现出来。如我所展望的那样一个系统可能成为证明这件事的一个方式，为各种各样的设计形式提供直接支持是它的明显应用。

把语法看成一个语言的用户界面，从这个观念很容易自然地演绎出有可能采用其他形式的用户界面。在系统里真正重要而不变的东西是语言的基本语义。任何时候都必须始终维持着这些东西，只要这样，如果我们真的需要具有熟悉的正文形式的传统C++代码，它总是可以生成出来的。

一个基于C++抽象表示的环境将使人能以另一种方式生产C++，以另一种方式观察C++。它也为连接、编译和执行代码提供了另一种方式。例如，连接可以在代码生成之前完成，因为这里将不需要产生目标代码以得到连接信息。解释器与编译器之间的差异将变成有些学术味道的东西了，因为它们都将依赖于具有大致相同形式的同一批信息。

9.4.5 小结

C++最有实力的地方并不是它在某个独到之处特别伟大，而在于它在事物的大范围变化中

都很不错。与此类似，从根本上说，发展并不是来自某个孤立的进步，而是来自在不同领域中的大量各种各样的进步。更好的库、更好的设计技术、接受过更好教育的程序员和设计师、语言标准、可选择的废料收集、对象通讯标准、数据库、基于非正文形式的环境、更好的工具、更快的编译等等，都将会有所贡献。

我认为，我们只是刚刚开始看到我们能够从C++中获得的效益。基础已经建立，但也不过是一个基础。面向未来，我期望能看到最主要的活动和进步能够从语言本身——这是一个基础——转移到依赖于它，在它上面构造起来的工具、环境、库和应用等方面去。

第二部分

第二部分描述在Release 1.0之后开发的C++特征。个别的特征将根据它们之间的逻辑关系结合成组，放进各章里。对于将语言看成一个整体而言，各种特征被引进C++的历史年表就不那么重要了，书中就没有反映这方面的情况。各章的排列并不太重要，它们可以按任何顺序阅读。这里描述的各种特征表明了C++的完成：从1985年开始设想，再经过这些年经验的锤炼。

章目录

- 第10章 存储管理
- 第11章 重载
- 第12章 多重继承
- 第13章 类观念的精炼
- 第14章 强制转换
- 第15章 模板
- 第16章 异常处理
- 第17章 名字空间
- 第18章 C预处理器

第10章 存储管理

有多少天赋
也打不败细节的纠缠。
——古语

对于小粒度存储分配和释放的需要——将存储分配与初始化分离——数组分配
——放置——存储释放问题——存储器耗尽——存储器耗尽的处理——自动废料收集

10.1 引言

C++为在自由存储区中分配存储提供了new运算符，也为释放存储提供了delete运算符（2.11.2节）。用户偶然也可能需要对于存储分配和释放的细粒度控制。

一类重要情况是对某个频繁使用的类做一个独立的类分配器（见[2nd, 第177页]）。许多程序中需要对某几个重要的类建立和删除大量的小对象，如树结点、链接表的链接、点、线、消息等等类的对象。采用一个通用分配系统做这种对象的分配和释放，很容易成为程序运行时间中的制约性因素，还可能主导了程序的存储需求。这里有两个因素在起作用：通用存储分配操作的运行时间和存储开销，以及由于各种大小的对象混合而产生的碎片问题。我发现，与不调整存储管理方式相比，引入一个专为特定类而做的分配程序，在典型情况下能使模拟器、编译系统、或者其他类似系统的速度加快一倍。我也看到过在碎片问题严重时加快十倍的情况。增加一个特定类分配程序，无论是自己手写还是取自某个标准库，在2.0版的特征中只不过是五分钟的额外工作。

需要细粒度控制的另一个例子是那种需要在资源非常紧张的环境里运行很长很长时间而又不能中断的程序。有严酷要求的实时系统常常需要有保证的、可以预期的存储获取，而且只能有最小的开销，因此也会提出类似的要求。传统上这类系统要完全避免动态存储分配。一个特定用途的分配程序可以用来管理这类的有限资源。

最后，我还遇到过几种情况，在那里某个对象必须放到某个特定的地址，或者放到某块特殊存储区域里，以满足硬件或者系统的特殊要求。

在Release 2.0中修改了C++的存储管理机制（2.11.2节），就是作为对这些需求的响应。在这里改进的主要是对分配的控制，还要依赖于程序员对所涉及问题的理解，期望程序员能结合使用语言的其他特征和技术，把通过某些细节方式完成的控制过程封装起来。这些机制在1992年完成，采用的方式就是引入了有关数组的operator new[]和operator delete[]。

在一些场合，来自Mentor Graphics的朋友提出了一些建议，他们正在用C++构造一个巨大而复杂的CAD/CAM系统。在这个系统里遇到了几乎所有已知的与程序设计规模有关的问题：数百个程序员、成百万行的代码、有严格的执行要求、严酷的资源限制、还有市场的最后期限。特别是Mentor的Archie Lachner对于存储管理问题提出了许多见解，在C++到2.0的转变中

起了很重要的作用。

10.2 将存储分配和初始化分离

在2.0以前，要做按类的存储分配和释放控制就必须通过对this的赋值（3.9节），这样做很容易出错，已经被声明为过时的方法了。在Release 2.0里提供了另一种方式，允许把有关分配的描述和初始化分开。原则上说，初始化由建构函数完成，这是在存储分配之后由独立的机制实施的。这样就容许使用多种不同的分配机制——其中某些可以是用户提供的。静态对象在程序连接时完成分配，局部对象在堆栈上分配，由new运算符创建的对象通过适当的operator new()分配。释放的处理方式与此类似。例如：

```
class X {
    // ...
public:
    void* operator new(size_t sz); // allocate sz bytes
    void operator delete(void* p); // free p

    X();           // initialize
    X(int i);     // initialize

    ~X();          // cleanup
    // ...
};
```

类型size_t是由实现定义的一个整数类型，用于保存对象的大小。它是从标准ANSI C里借来的。

运算符new的职责是保证互相分离的存储分配和初始化能正确地放在一起使用。例如，编译系统的一个工作是产生对分配程序的一个调用X::operator new()，并在为X所用的new中产生一个对X的建构函数的调用。从逻辑上看，X::operator new()是在建构函数之前调用的，因此它必须返回一个void*，而不是X*。对应的建构函数将在分配给它的存储之上构造起一个X对象。

与此相对应，析构函数“消解”掉一个对象，给operator delete()留下一块没有特征的存储区，让它去释放。因此operator delete()以一个void*为参数，而不是X*。

这种普遍规则也同样适用于有继承的情况，因此，一个派生类的对象将会用基类的operator new()进行分配：

```
class Y : public X { // objects of class Y are also
                    // allocated using X::operator new
    // ...
};
```

为此，这里的X::operator new()同样需要一个参数，描述所要求分配的存储量，因为一般说sizeof(Y)与sizeof(X)并不一样大。不幸的是，新用户们常常感到很困惑，为什么他们必须声明这个参数，而在调用时又不必去显式地提供这个信息。这里采用的概念是让用户声明一个带参数的函数，而让“系统”去“神秘地”提供这个参数。看起来有些人很难把握住这种概念。这个修改增加了复杂性，但是我们取得了让基类为一集派生类提供分配和释放服务的能力，以及更规范的继承规则。

10.3 数组分配

一个类所特定的`X::operator new()`仅仅用在类`X`的独立对象方面（包括用于那些由`X`派生的，又没有自己的`operator new()`的类），这也就意味着

```
X* p = new X[10];
```

与`X::operator new()`没有关系，因为`X[10]`是一个数组而不是类`X`的一个对象。

这又引起了一些抱怨，因为不能允许用户对`X`的数组的分配加以控制。无论如何，我曾经坚持认为一个“`X`的数组”不是一个`X`，因此不能使用`X`的建构函数。因为如果想让它能用到数组上，那么写`X::operator new()`的人就必须“按照这种情况”处理数组分配的问题，这会使重要的普遍性情况变得更复杂：如果情况并不很重要，那为什么还要考虑特殊的分配程序呢？我还指出，只控制像`X[d]`这样的一维数组还不够，像`X[d1][d2]`这样的二维数组又该怎么办呢？

然而，没有数组分配的控制机制确实也在一些实际情况中造成了许多困难局面，最后标准化委员会提出了一种解决方案。这里最关键的问题是，没有任何办法阻止用户在自由空间里分配数组，甚至没有办法去控制它。在那些依赖于逻辑上不同的存储管理模式的系统里，这很可能带来极其严重的问题，因为用户很可能就简单地把很大的动态数组放置在默认的分配区域里。我原来并没有认识到这个问题蕴涵着什么。

```
class X {
    // ...
    void* operator new(size_t sz);    // allocate objects
    void operator delete(void* p);

    void* operator new[](size_t sz); // allocate arrays
    void operator delete[](void* p);
};
```

所采纳的解决方案很简单，专门提供一对函数特别处理数组的分配和释放问题：

数组分配程序用于为任何维数的数组获取空间。与所有的分配程序一样，`operator new[]`的工作就是提供所需要的那么多字节的存储，它本身并不关心这些存储将被如何使用。特别是它不需要知道数组的维数或者元素个数。Mentor Graphics的Laura Yaker是引进这种数组分配和释放程序的最早的推动者。

10.4 放置

两个相关的问题用一种通用机制解决了：

- 1) 我们需要一种机制把对象安放到某个特定地址。例如，把一个表示进程的对象放到特定硬件所要求的特定地址去。
- 2) 我们需要一种机制在某个特定分配区里分配对象。例如在一个多处理器系统的共享存储中，或者从由某个特定对象管理器控制的区域中分配对象。

解决方案是允许对`operator new()`进行重载，以便为`new`运算符提供一种允许有附加参数的语法形式。例如，一个能把对象分配到特定分配区的`operator new()`可能被定义为下面这种样子：

```
void* operator new(size_t, void* p)
{
    return p; // place object at 'p'
}
```

它的调用可能是：

```
void* buf = (void*)0xF00F; // significant address
X* p2 = new(buf)X; // construct an X at 'buf'
                    // invokes: operator new(sizeof(X),buf)
```

由于其使用方式，这种为operator new()提供附加信息的语法形式被称为放置语法形式。请注意，每个operator new()调用总是以被分配的存储大小作为它的第一个参数，这是根据被分配对象而自动提供的。

说起来，我在那时实际上也低估了放置问题的重要性。有了放置机制之后，运算符new就不再只是一种简单的存储分配机制，因为我们可以给特定的存储位置关联上任意的逻辑性质，这就使new能够起一种通用资源管理的作用。

为某个特定分配场地定义的operator new()可能具有下面样子：

```
void* operator new(size_t s, fast_arena& a)
{
    return a.alloc(s);
}
```

其使用像是：

```
void f(fast_arena& arena)
{
    X* p = new(arena)X; // allocate X in arena
    // ...
}
```

在这里的fast_arena也假定是一个类，它有一个成员函数alloc()，可用于获得存储。例如有：

```
class fast_arena {
    // ...
    char* maxp;
    char* freep;
    char* expand(size_t s); // get more memory from
                           // general purpose allocator
public:
    void* alloc(size_t s) {
        char* p = freep;
        return ((freep+=s)<maxp) ? p : expand(s);
    }
    void free(void*) {} // ignore
    clear(); // free all allocated memory
};
```

这应该是一个特定的分配场地，处理快速分配并几乎总是立即释放的情况。分配区的一种重要应用是提供特定的存储管理语义。

10.5 存储释放问题

在operator new()和operator delete()之间有一种明显的、经过深思熟虑的不

对称性。前者能够被重载而后者不能。这与建构函数和析构函数之间的不对称也是互相匹配的。因此，在构建对象的时候你可能可以在四个分配器和五个建构函数中做选择，而当它到了被销毁的时候，则只存在着惟一的一种选择：

```
delete p;
```

这样做的理由是，在建构对象的那一点，在原则上你知道所有的东西，而当到了删除它的时候你剩下的不过是一个指针，这个指针可以正好是也可以不是该对象的类型。

当用户通过基类的指针去删除一个派生类的对象时，为了使析构操作能够正常完成，关键就是要使用一个虚的析构函数：

```
class X {
    // ...
    virtual ~X();
};

class Y : public X {
    // ...
    ~Y();
};

void f(X* p1)
{
    X* p2 = new Y;
    delete p2;      // Y::~Y correctly invoked
    delete p1;      // correct destructor
                    // (whichever that may be) invoked
}
```

这样做就能保证，如果在类层次中存在着一个局部的operator delete()函数，正确的函数就一定会被调用到。如果没有使用虚的析构函数，那么Y的析构函数所描述的清理动作就不会被执行。

此外，在这里并不存在一种语言特征，使得我们能像分配函数的选择机制那样去选择释放函数：

```
class X {
    // ...
    void* operator new(size_t); // ordinary allocation
    void* operator new(size_t, Arena&); // in Arena

    void operator delete(void*);
    // can't define void operator delete(void*, Arena&);
};
```

问题仍然是删除的点，我们不能期望用户知道有关对象是如何分配的。当然，最理想的情况是用户根本就不必去释放对象，这也是特殊分配场地的一种用途。也可以定义一种分配场地，令它在程序执行的某个特定点以整体的方式进行释放，或许还可以为某个分配场地定义一个特定的废料收集系统。前一种做法是相当常见的，后一种则不很常见，要做就需要仔细做好，那才可能超过标准的可装入的保守式废料收集系统[Boehm, 1993]。

更常见的情况是编好函数operator new()，让它在对象里留下一个指示，以使对应的

`operator delete()`能够知道这些对象应该如何释放。请注意，这也是在做存储管理，与由建构函数建立和析构函数销毁的对象相比，这是在更低的概念层次上做。因此，包含这种信息的存储位置不应该在实际对象的内部，而应该在与之相关的某个地方。例如某个`operator new()`可能把存储管理信息放在作为它返回值的指针再前面的一个字里。作为另一种方式，`operator new()`也可以把信息存放到另一个位置，使建构函数和其他函数能够找到，从而确定是否已经在某个自由存储里分配了一个对象。

不允许用户对`delete`进行重载是不是一个错误？如果是，那么保护人们以提防他们自己就是一种误导。我无法做出决定，但是我也十分确信这是个很难对付的情况，采取任何解决办法都会引出许多问题。

Release 2.0引进了显式调用析构函数的可能性，以对付某些不大常见的情况，在那里存储分配和释放是相互完全分离的。一个实际例子是某种容器，它需要为自己所包含的对象完成所有的存储分配工作。

数组的释放

C++从C那里继承了一个问题：指针可以指向个别对象，而这个对象实际上却是某个数组的初始元素。一般说编译系统无法告诉我们这一点。当一个指针指向某数组的第一个元素时，我们常称它是指向了这个数组，而数组的分配和释放都是通过这种指针进行的。例如：

```
void f(X* p1)    // p1 may point to an individual object
                  // or to an array
{
    X* p2 = new X[10]; // p2 points to the array
    // ...
}
```

我们怎样才能保证数组能够被正确地删除？特别是，我们怎么才能保证对数组的所有元素都调用了相关的析构函数？Release 1.0对这个问题没有提供满意的回答，Release 2.0为此引进了一个显式的数组删除运算符`delete[]`：

```
void f(X* p1)    // p1 may point to an individual object
                  // or to an array
{
    X* p2 = new X[10]; // p2 points to the array
    // ...
    delete p2;        // error: p2 points to an array
    delete[] p2;      // ok
    delete p1;        // maybe ok, trust the programmer
    delete[] p1;      // maybe ok, trust the programmer
}
```

这样一来，普通的`delete`就不再需要同时处理个别对象和数组两种情况了，这避免了将复杂性引进分配和释放个别对象的一般情况中，也使个别对象不再需要附带那些专门为数组释放而用的信息。

`delete[]`的某个中间版本曾要求程序员明确写出数组元素的个数。例如：

```
delete[10] p2;
```

这种方式被证明是很容易引进错误的。因此，维护数组元素个数的负担后来就被转到了语言

的实现方面。

10.6 存储器耗尽

要找某种需要的资源而又无法得到，这是一个普遍的而又很难对付的问题。我曾经确定（在2.0之前）异常处理是一个方向，应该到那里去寻找针对这类问题的一个具有普遍意义的解决方法（3.5节）。但异常处理（第16章）在那时还是很远的将来的事情，而自由存储耗尽这个特殊的问题是无法等待的。在这个中间阶段的几年里，还是需要有某种解决方案，即使是一个丑陋的方案。

立即需要解决的问题有两个：

- 1) 在任何情况下，当一个库调用因为存储耗尽而失败时（更一般的，在任何库调用失败时），用户都应该能够取得控制。这对于AT&T内部的用户而言是一个绝对的要求。
- 2) 普通用户不需要在每次分配操作之后去测试存储耗尽的情况。从C得到的经验告诉我们，用户一般都不去做这种测试，即使是人们认为应该这样做。

对于第一个问题的处理要求当operator new()返回0时建构函数根本就不执行。在这种情况下，new表达式本身也应该产生0值。这就使关键性的软件在出现存储分配问题时能够保护自己。例如：

```
void f()
{
    X* p = new X;
    if (p == 0) {
        // handle allocation error
        // constructor not called
    }
    // use p
}
```

满足第二个需要的方法是用一个称为new_handler的东西，它是一个由用户提供的函数，如果运算符不能找到存储，就保证去调用这个函数。例如：

```
void my_handler() { /* ... */ }

void f()
{
    set_new_handler(&my_handler); // my_handler used for
                                  // memory exhaustion
                                  // from here on
    // ...
}
```

这种技术在[Stroustrup, 1986]中提出，已经成为处理资源需求偶然失败时被普遍采用的一种模式。简单地说，new_handler可以：

- 寻找更多的资源（也就是说，去寻找更多的自由存储）；或者
- 产生一个错误信息，并退出（也是个办法）。

有了异常处理，“退出”可能比直接终止程序稍微平和一点（16.5节）

10.7 自动废料收集

我有意地这样设计C++, 使它不依赖于自动废料收集(通常就直接说废料收集)。这是基于自己对废料收集系统的经验, 我很害怕那种严重的空间和时间开销, 也害怕由于实现和移植废料收集系统而带来的复杂性。还有, 废料收集将使C++不合适做许多低层次的工作, 而这却正是它的一个设计目标。我喜欢废料收集的思想, 它是一种机制, 能够简化设计、排除掉许多产生错误的根源。但是我也确信, 如果原来就把废料收集作为C++的一个有机的组成部分, 那么C++可能早就成为死胎了。

我过去的观点是, 如果你需要废料收集, 你或者可以自己实现某种自动存储管理模型, 或者是去使用某种直接支持它的语言, 例如我个人始终喜爱的Simula。今天这个问题就不能划分得那样清楚了。有许多可以实现或者移植的软件资源; 存在着许多无法简单地改写成其他语言的C++软件; 废料收集系统也有了很大改进, 存在许多新技术可以用到“家酿”的废料收集系统中, 而按照我原来的想象, 这种东西是无法从个别的项目提升到通用库的。更重要的是, 今天人们用C++做的项目更雄心勃勃, 有些项目可能从废料收集中获益, 也能够接受它的开销。

10.7.1 可选的废料收集

按照我现在的认识, 可选的废料收集对于C++是一条合适的路。究竟应该怎样做, 现在还不是很清楚。但是我们已经有了一些看法, 今后几年可能看到几种形式的东西。已经存在一些实现, 剩下的问题只不过是将它们从研究阶段转化到产品代码的时间。

需要废料收集的基本理由是很容易陈述的:

- 1) 对于用户而言, 有废料收集是最方便的。特别是它能使一些库的构造和使用更加简单。
- 2) 对于许多应用而言, 废料收集比用户提供的存储管理模式更可靠。

反对理由可以多列出几条, 但它们都不是最根本的, 而是关于实现和效率方面的:

- 1) 废料收集带来运行时间和空间的开销, 对于在目前计算机硬件上运行的许多现在的C++应用而言, 这是无法负担的。
- 2) 许多废料收集技术隐含着服务的中断, 对于一些很重要的应用类, 例如有严格时间要求的实时应用、设备驱动程序、在较慢的硬件上的人机界面代码、操作系统核心等, 这种情况是无法接受的。
- 3) 有些应用系统并没有传统通用计算机上的那些硬件资源。
- 4) 有些废料收集方案需要禁止某些基本的C机制, 例如指针算术、不加检查的数组、不加检查的函数参数(如printf()所用的参数机制)等。
- 5) 有些废料收集方案提出了一些对于对象布局和对象创建的限制, 这些都会使与其他语言的接口变得更复杂。

我还知道许多支持的和反对的意见, 但是不需要列出更多的东西了。已经有了充分多的论据反对另一种观点: 每个应用在有了废料收集之后会做得更好些。类似地, 也有充分的论据反对对方的观点: 没有应用可能因为有了废料收集而做得更好。

在对有废料收集和没有废料收集的系统做比较时, 应该记住: 并不是每个程序都需要永无休止地运行下去; 并不是所有的代码都是基础性的库代码; 对于许多应用而言, 出现一点

存储流失是可以接受的；许多应用可以管理自己的存储，而不需要废料收集或者其他与之相关的技术，如引用计数等。C++需要废料收集的方式与那些没有真正局部变量（2.3节）的语言完全不同。当存储管理的行为方式比较规范时，可以通过不那么具有一般性的方式来处理（例如通过专用的分配器（10.2节，10.4节，15.3.1节，[2nd, 5.5.6节，13.10.3节]）、自动存储或静态存储（2.4节））。与手工的或自动的通用废料收集系统相比，这些方式一定能够获得显著的速度和空间优势。对于许多应用而言，这种优势是极其重要的，这将使自动废料收集系统给其他应用带来的利益在这里显得无关大局。而在一个理想的实现中，这种优势将不会受到废料收集系统存在的拖累。实际上，那个收集系统或者根本就不会被调用，或者极少被调用，以至对大部分应用的整体效率几乎没有影响。

我的结论是，从原则上和可行性上说，废料收集都是需要的。但是对今天的用户、普遍的使用和硬件而言，我们还无法承受将C++的语义和它的基本库定义在废料收集系统之上的负担。

实际问题是，可选的废料收集能否成为C++的一个有活力的观点？当废料收集可以用的时候（不是“如果……”），我们将有两种方式去写C++程序。这在原则上并不比管理几个不同的库、几个不同的应用平台等等更困难，这些事情我们已经做得很多了。要采用某种广泛使用的通用程序设计语言，就必须做出这类的选择。不可能要求任何C++程序的执行环境总是一模一样的，无论它是运行在导弹头里面，或者SRL照相机里，或者PC里，或者电话交换机里，或者某种UNIX系统里，或者IBM主机里，或者Mac计算机里，或者超级计算机里，或者其他什么地方。如果能够以某种合理的方式提供，废料收集将能成为人们为一个应用系统选择运行环境时的一个选择项。

如果不把它作为C++标准的组成部分，废料收集也可能成为合法的和有用的东西吗？我是这样认为的。再说，我们也没有在标准里描述选择性的废料收集，这是因为我们还没有一种从某些方面看能够接近成为标准的模型。一种试验性的模型必须经过足够多的、范围足够广泛的实际应用的检验。还有就是它必须没有不可避免的缺陷，不能使C++在重要的应用领域成为不能接受的东西。有了这些成功的经验之后，实现者们将会努力奋斗，提供最好的实现。我只能希望他们不要选择互不相容的模型。

10.7.2 可选的废料收集应该是什么样子的

这里存在着若干选择，因为对基本问题有几种不同的解决方法。一个想法是尽可能扩大能够同时在有废料收集和没有废料收集的环境中运行的程序的范围。对于实现系统的人、库的设计者和应用程序员来说，这也是一个很重要的很吸引人的目标。

理想情况是，带有废料收集的系统实现应该有这样一种性质：当你不使用废料收集时，它应该工作的像没有废料收集的系统一样好。如果程序员必须说：“这个程序不需要用废料收集功能”，那么这个目标是很容易达到的。但如果要求这个实现准备做废料收集，而又能通过调整自己的行为，使之能达到没有废料收集的实现那样的性能，事情就会变得极端困难了。

与此相反，废料收集系统的实现者可能需要从用户得到某些“提示”，以便使程序的执行达到可以接受的水平。例如，一种方式可能是要求用户说明哪些对象需要做废料收集，哪些对象不需要做（例如，这些东西来自根本没有废料收集的C或者Fortran库）。只要有可能，不带废料收集的实现就会忽略这些提示。另一种方式是直接将它们从程序里去掉，这也非常

简单。

有些C++操作，例如病态的类型强制转换（casting）、指针与非指针的联合、指针算术等等，都是对废料收集严重不利的东西。对于写得很好的C++代码而言，这些操作一般不常使用，所以出现了一些禁止它们的尝试。两种思想在这里又发生了冲突：

- 1) 禁止这些不安全操作：这将使程序设计更安全，也使废料收集的效率更高。
- 2) 不要禁止任何在今天合法的C++程序。

我认为还是可以达成一种妥协。我想可能调制出一种废料收集模型，它能够应付（几乎）所有的合法C++程序，而对那些没有不安全操作的程序，它能够工作得更有效。

在实现某种废料收集模型时，人们必须确定是否需要对被收集的对象调用有关的析构函数。想确定怎样做才算正确并不是一件很容易的事情。在[2nd]里我写到：

“废料收集可以被看成是在一个有限的存储中模拟无限存储的一种方法。把这件事情记在心里，我们就可以回答一个常见的问题：在废料收集回收一个对象时，它应该调用其析构函数吗？回答是不，因为被放回自由存储的这些对象并没有经过删除，因而绝不会被销毁。根据这个认识，使用`delete`就是直接要求调用析构函数（同时也是通知系统，说这个对象的存储可以重新再用了）。但是如果确实希望对于从自由存储分配之后没有删除的对象执行某种操作，那么又该怎么办呢？请注意，对于静态的和自动的对象都不会出现这个问题，因为总会隐式地对它们调用析构函数。还应该注意到，要“在废料收集时”执行动作，其执行时间是无法预计的，原则上说，这种动作可能在对于这个对象的最后访问直到“整个程序终止”之间的任何时刻发生。这也就意味着，在它们的执行期间，程序的状态是无法知道的。这种情况也进一步说明，对这样的动作很难写出正确的程序，它们也不像人们设想的那样有用。”

如果在某个地方真的需要这种动作，需要在某个无法说清楚的“析构时刻”做点事情，这个问题可以通过提供一个注册服务程序的方式解决。如果一个对象希望在“程序结束的时候”执行某种动作，那么就吧它的地址和一个到对应“清理”函数的指针放进一个全局性的关联数组里。”

现在我不那么确定了。虽然这种模型能够工作，但是或许让废料收集程序去调用析构函数会更简单些，也值得采纳。这完全依赖于被收集的对象是什么，在它们的析构函数中要做什么事。这是一个问题，不能通过纯理论的方式去决定，在其他语言的经验中也没有看到什么有关的东西。不幸的是它还是一个很难导出真实试验的问题。

我不存在任何幻想，以为给C++构造出一个可以接受的废料收集机制是一件很容易的事情——我仅仅是不认为这一定做不到罢了。因此，只要有很多人去研究这个问题，不久就会有一些解决办法浮现出来。

第11章 重 载

魔鬼隐藏在细节之中。

——古语

细粒度重载解析——歧义控制——空指针——类型安全的连接——名字压延——控制复制、分配、派生等——灵巧指针——灵巧引用——增量和减量——指数运算符——用户定义运算符——组合运算符——枚举——布尔类型

11.1 引言

运算符就是为了提供使用的方便性。考虑简单公式 $F=M*A$ ，没有一本基础物理书上将它写成`assign(F, multiply(M, A))`^Θ。当变量可以具有不同类型时，我们就必须确定是允许混合算术呢，还是要求显式地把运算对象转换到共同的类型。例如，如果M是`int`而A是`double`时，我们或者是接受 $M*A$ 并推断说M必须在作乘法之前提升到一个`double`，或者可以要求程序员写某种东西，例如`double(M)*A`。

由于选择了前者——与C、Fortran和其他所有被广泛用于计算领域的语言一样——C++就进入了一个不存在完美解决方案的困难领域。从一方面看，人们希望有“自然的”转换而不出现编译系统的唠唠叨叨；在另一方面他们又不希望感到意外。对于什么是自然，不同的人会有全然不同的看法，至于人们能够容忍哪些意外，情况也差不多。这些，再加上要受C语言中已经相当混乱的内部类型和转换的兼容性的约束，结果就产生了一个从本质上就极难解决的问题。

对于表达方式灵活和自由的欲望，与对安全性、可预见性以及简单性的追求是互相冲突的。这一章将考察由这种冲突引起的对于重载机制的精炼过程。

11.2 重载的解析

从一开始，函数名和运算符的重载就被引进了C++（3.6节）[Stroustrup, 1984b]，后被广泛使用，但是与重载机制有关的问题也逐渐浮现出来。关于Release 2.0所做的改进总结在[Stroustrup, 1989b]里：

“对C++的重载机制做了些修改，以便能将过去认为是‘太类似’的类型被解析出来，还取得了对声明顺序的无关性。作为结果的这个模式将更具表达力，也能捕捉到更多的歧义性错误。”

在细粒度解析方面的工作使我们得到了基于`int/char`、`float/double`、`const/非const`以及基类/派生类进行重载的能力。顺序无关性则清除掉许多难对付的错误的一个根源。在下面，我将考察重载的这两个方面情况。最后我将解释为什么把`overload`关键字作为过

^Θ 有些人甚至偏爱 $F=MA$ ，但是解释清楚如何可能采用这种东西（“没有空格的重载”）已经超出了本书的范围。

时的东西。

11.2.1 细粒度解析

在开始设计时，C++的重载规则接受了C语言内部类型的限制 [Kernighan, 1978]。也就是说，在这里没有float类型的值（从技术上说，没有对应的右值），因为对一个float的计算将立即被扩展为一个double。与此类似，在这里也没有类型为char的值，因为char的每个使用都被扩展为一个int。这引起了许多关于无法自然地提供单精度浮点库的抱怨，也使字符操作函数不必要地非常容易出错。

考虑一个输出函数。如果我们无法基于区分char/int做重载，我们就必须使用两个不同的名字。事实上开始的流库（8.3.1节）用的就是：

```
ostream& operator<<(int); // output ints (incl. promoted
                           // chars) as sequence of digits.
ostream& put(char c);    // output chars as characters.
```

但是，许多人写：

```
cout<<'X';
```

并（很自然）吃惊地发现，在他们的输出中出现的是88（ASCII 'X'的数值）而不是字符X。

为了克服这个问题，后来对C++的类型规则做了修改，对类型char和float等的未提升形式也能使用重载解析机制。进一步说，文字的字符如'X'被定义为具有类型char。与此同时，还采纳了那时最新发明的ANSI C有关表达unsigned和float字面量的记法，所以我们就可以写：

```
float abs(float);
double abs(double);
int abs(int);
unsigned abs(unsigned);
char abs(char);

void f()
{
    abs(1);      // abs(int)
    abs(1U);     // abs(unsigned)
    abs(1.0);    // abs(double)
    abs(1.0F);   // abs(float)
    abs('a');    // abs(char)
}
```

在C语言里，字符字面量（例如'a'）的类型是int。令人吃惊的是，在C++里给'a'指定类型char居然没有造成什么移植性问题。除了病态的例子sizeof('a')，其他能在C和C++里表述的每种结构都会给出同样结果。

在把字符字面量的类型定义为char时，我部分地依靠了来自Mike Tiemann的一个报告，他描述了用一个编译选项在GNU的C++编译器里提供这种解释的经验。

与此类似，人们也发现利用const和非const之间的差异也能产生好的效果。基于const重载的一个重要应用就是提供一对函数：

```
char* strtok(char*, const char*);
```

```
const char* strtok(const char*, const char*);
```

用它们作为下面ANSI C标准函数的另一种选择：

```
char* strtok(const char*, const char*);
```

在C语言里，`strtok()`返回作为其第一个参数传递进去的`const`串的一个子串。C++的标准库将不允许这个子串不是`const`，因为不能接受隐含地违背类型系统的情况。而在另一方面，又需要尽可能地减少与C语言的不兼容性。提供两个`strtok`函数就能够允许`strtok`的大部分合理使用了。

允许基于`const`的重载，也是收紧有关`const`的规则，以及倾向于强制性地要求这一规则的工作的一部分（13.3节）。

经验说明，在函数匹配时，还应该考虑通过公用类派生建立起来的分层结构的情况，以便使存在选择的时候，到“最终的派生类”的转换能够被选中。仅在没有其他指针参数能够匹配时才会选中`void*`参数。例如：

```
class B { /* ... */ };
class BB : public B { /* ... */ };
class BBB : public BB { /* ... */ };

void f(B*);
void f(BB*);
void f(void*);

void g(BBB* pbbb, BB* pbb, B* pb, int* pi)
{
    f(pbbb); // f(BB*)
    f(pbb); // f(BB*)
    f(pb); // f(B*)
    f(pi); // f(void*)
}
```

这个歧义消解规则与虚函数调用规则完全匹配，在那里被选中的也是最后派生类的成员函数。本规则的引入清除了这样一个错误根源。这个修改实在太明显了，人们用打哈欠的方式来表示同意（“你的意思是说原来不是这个样子吗？”）。一个错误消失了，这就是全部的故事。

这个规则还有一个有趣的性质，也就是说，它将`void*`确立为类转换树的根。这件事与下面的观点很相配：建构函数从原始存储中做出对象，析构函数完成反向的过程，从对象里制造出原始存储（2.11.1节与10.2节）。例如将`B*`转换到`void*`就使一个对象可以被看成原始的存储，在其中没有任何使人感兴趣的性质。

11.2.2 歧义控制

原来的C++重载规则根据声明的顺序去消解歧义性。声明将被顺序地进行试验，第一个能够匹配的东西“取胜”。为了使这种方式能够说得过去，在匹配过程中只能接受非缩窄的转换。例如：

```
overload void print(int); // original (pre 2.0) rules:
void print(double);

void g()
```

```

{
    print(2.0); // print(double): print(2.0)
    // double->int conversion not accepted.
    print(2.0F); // print(double): print(double(2.0F))
    // float->int conversion not accepted
    // float->double conversion accepted.
    print(2); // print(int): print(2).
}

```

这个规则很容易表述，也易于为用户所理解，编译时很有效，实现者也很容易把它做正确。但是，它还成为了错误和混乱的不尽之源。因为调换一些声明的顺序就可能完全改变一段代码的意义：

```

overload void print(double); // original rules:
void print(int);

void g()
{
    print(2.0); // print(double): print(2.0).
    print(2.0F); // print(double): print(double(2.0F))
    // float->double conversion accepted.
    print(2); // print(double): print(double(2))
    // int->double conversion accepted.
}

```

简而言之，顺序依赖性极易产生错误。它已经成为一个严重的障碍，阻挡了使C++程序设计向更广泛地使用库的方向发展的努力。我的目标是转移到一种新的观点去，将程序设计看成是从独立的程序片段组合出程序的过程（11.3节），而顺序依赖性就是许多障碍中的一个。

这里的暗礁是，与顺序无关的重载规则将使C++的定义和实现复杂化，因为需要在最大程度上保持与C语言以及原来的C++的兼容性。特别是最简单的规则：“如果一个表达式存在两个可能的合法解释，它就是歧义的也是非法的”并不是一种可能选择。例如，按照这个规则，上面例子里所有的print()调用就都是歧义的和非法的了。

我当时总结出，我们需要某种“更好匹配”规则的概念，这将使我们认为一个精确的匹配胜过一个需要转换的匹配，一个安全转换的匹配（例如从float到double）胜过一个不安全的（缩窄的、破坏值的等等）转换（例如从float到int）。这引起了一系列的讨论、精练和重新考虑，经历了若干年时间。有些细节仍然还在标准化委员会里讨论着，主要的参加者包括Doug McIlroy、Andy Koenig、Jonathan Shapiro和我。很早Doug McIlroy就指出，我们已经极其冒险地接近了试图为隐式转换设计一套“自然”系统的问题。他考虑过PL/I的规则，那是他帮助设计的，他还证明了，对于一个丰富的通用数据类型集合，不可能设计出这样的“自然”系统。而C++却提供了一集很丰富的带着混乱转换的内部类型，再加上在用户定义类型之间定义任意转换的能力。我对于进入这个沼泽所陈述的理由是：我们现在别无选择，只能去试一试。

与C语言的兼容性，人们的期望，以及允诺用户定义的类型可以像内部类型一样使用的目标等等都不允许我们禁止隐式的类型转换。回过头再看，我还是赞成将隐式转换推向前进的决策，我也赞成Doug的观点，要把隐式类型转换所导致的使人惊异的东西减到最少，这里有本质性的困难，这种使人惊异的东西是不可能完全消除的（至少，在保留了与C兼容性的前提

之下)。不同的程序员会有不同的预期,因此,无论你选择什么规则,也总会有人在某个时候感到吃惊。

一个最根本的问题是,在内部类型的隐式转换图中存在着环路。例如,不仅存在从char到int的隐式转换,也存在从int到char的转换。这就出现了产生永无休止的微妙错误的潜在可能性,阻止我们采纳基于一个转换格的隐式转换模型。与此相反,我们需要谋划出一个在函数声明确定的类型与实际参数的类型之间进行“匹配”的系统。这个匹配应该包含了我们认为比其他东西更少出错、更少引起惊异的那些转换。在这里,还需要符合C的标准提升和标准转换规则。我把这个模型的2.0版本描述为[Stroustrup, 1989b]:

“这里是对新规则的一个稍微做了一点简化的解释:请注意,除了极少数的例外情况,在老规则允许顺序依赖性的地方,新规则是与之兼容的;老程序在新规则下也能给出完全相同的结果。最近两年左右的C++实现就已经开始对在这里‘违法’的顺序依赖性解析问题提出警告信息了。

C++区分了5种‘匹配’:

- 1) 匹配中不转换或者只使用不可避免的转换(例如,从数组名到指针,函数名到函数指针,以及T到const T)。
- 2) 使用了整数提升的匹配(像ANSI C标准所定义的。也就是说从char到int、short到int以及它们对应的unsigned类型)以及float到double。
- 3) 使用了标准转换的匹配(例如从int到double、derived*到base*、unsigned int到int)。
- 4) 使用了用户定义转换的匹配(包括通过建构函数和转换操作)。
- 5) 使用了在函数声明里的省略号...的匹配。

让我们考虑只有一个参数的函数,这里的想法是总选择‘最好的’匹配,也就是说在上面表里最高的匹配。如果存在着两个最好匹配,这个调用就是有歧义的,将产生一个编译错误。”

上面的例子说明了这个规则。有关规则的另一个更精确的版本可以在ARM里找到。

还需要有另一个规则去处理多于一个参数的函数[ARM]:

“对涉及多于一个参数的调用,一个函数要想被选中,那就要求它至少在某一个参数上比其他任何函数都匹配得更好,而对每个参数都至少与其他函数匹配得同样好。例如:

```
class complex {
    // ...
    complex(double);
};

void f(int,double);
void f(double,int);
void f(complex,int);
void f(int ...);
void f(complex ...);

void g(complex z)
```

```

{
    f(1,2.0);    // f(int,double)
    f(1.0,2);    // f(double,int)
    f(z,1.2);    // f(complex,int)
    f(z,1.3);    // f(complex ...)
    f(2.0,z);    // f(int ...)
    f(1,1);      // error: ambiguous,
                  // f(int,double) or f(double,int) ?
}

```

在第三个和倒数第二个调用里，从double到int的不幸的窄转换将导致警告。这种窄转换被允许，以便保持与C语言的兼容性。在这儿的特殊情况下，这种窄转换没什么大害，但是在许多情况下从double到int将导致值的破坏，绝不能随意使用。”

有关多参数情况的这些规则经过了仔细的推敲和形式化后，派生了“交叉规则”[ARM, 第312~314页]。这个交叉规则最先由Andrew Koenig在与Doug McIlroy、Jonathan Shopiro和我的讨论中提出。我相信Jonathan就是那个发现了极其古怪的例子的人，这些例子证明了这个规则的必要性[ARM, 第313页]。

请注意这里对兼容性的考虑有多么重要。我的观点是，如果考虑得稍微少一点，当前和将来的C++程序员的绝大多数就会有严重反响。一个更简单、更严格、更容易理解的语言除了应该能吸引那些对现存语言一直不满的程序员外，还必须能吸引更多敢于冒险的程序员。如果我们所做的那些设计决策一直系统地将简单和优雅置于兼容性之上的话，今天的C++必然会更小，也会更清晰，但它也将会是一个很不重要的异教语言。

11.2.3 空指针

什么是表述一个指针没有指向任何对象（空指针）的正确方式，这个讨论可能是引起激烈争论最多的东西了。C++从经典C语言[Kernighan, 1978]继承了它的定义：

“一个能求出0值的常量表达式被转换为一个指针，通常称为空指针。这个值所产生的指针将保证能与任何对象或者函数的指针互相区分。”

在ARM里进一步警告说：

“请注意，空指针并不一定用与整数0同样的二进制模式表示。”

这个警告是对一种常见误解的反应：许多人认为，既然 `p = 0` 给指针 `p` 赋了空指针值，那么空指针的表示必然就和整数0完全一样，就是说，是一个全0的二进制模式。实际并不是这样。C++是具有足够强的类型的语言，这样，一个像空指针这样的概念完全可以采用实现者选定的任何方式表示，与这个概念在正文形式中的表示没有关系。只有一个例外，那就是在人们用省略号抑制了对函数参数的检查时：

```

int printf(const char* ...); // C style unchecked calls
printf(fmt, 0, (char)0, (void*)0, (int*)0, (int(*)())0);

```

在这里所有的强制转换都是必需的，这样才能描述到底需要哪一种0。在这个例子里，可以想象是传递了五个不同的值。

在K&R C里函数的参数根本不检查，即使在ANSI C里你也还不能依赖于参数检查，因为它是可选的。由于这个原因，也因为0在C和C++里难以分辨，还因为在其他语言里人们习惯了用一个符号常量来表示空指针，C程序员就倾向于用一个称为NULL的宏表示空指针。不幸的是，在K&R C里并不存在对NULL的可移植的正确定义。在ANSI C里(void*)0是NULL的一个合理的且越来越被公众接受的定义。

但是(void*)0却不是在C++里对空指针的一个好选择：

```
char* p = (void*)0; /* legal C, illegal C++ */
```

不经过强制转换，void*就不能对任何东西赋值。允许将void*隐式地转换到任意指针类型，将会在类型系统上打开一个大洞。当然可以把(void*)0作为一个特殊情况，但应该只在极端需要时才值得接纳特殊情况。还有，C++的使用是在ANSI C标准存在之前很长时间，我也极不希望C++的关键部分依赖于一个宏（第18章）。因此我就用一个普通的0，这么多年用它也工作得很好。强调用符号常量的人们通常定义下面两者之一：

```
const int NULL = 0; // or
#define NULL 0
```

如果只考虑编译程序的问题，在此之后NULL和0实际上就是同义词了。不幸的是，许多人已经在他们的代码里加上了NULL、NIL、Null、null等等的定义，再提供另一个定义将是非常冒险的。

如果用0（无论如何拼写）表示空指针，那就会出现一类无法捕捉到的错误。考虑：

```
void f(char*);
void g() { f(0); } // calls f(char*)
```

现在加上另一个f()，g()就会不动声色地改变了意义：

```
void f(char*);
void f(int);
void g() { f(0); } // calls f(int)
```

因为在这里用的0是一个int，又可以提升到一个空指针，这是没用直接描述的空指针而产生的一个很不幸的副作用。我想一个好的编译系统应该给出警告，但是在做Cfront时还没有想到这个情况。认为这里的f(0)存在歧义；而不是按更偏向f(int)方式解析也可以行得通，但这样做可能不会使那些希望NULL或nil有魔力的人感到满意。

经过comp.lang.c++和comp.lang.c上一场常见的热战之后，我的一个朋友认识到：“如果0就是他们最糟的问题，那么他们真是太幸运了。”按照我的经验，用0表示空指针在实践中并不是一个问题。我始终觉得有关的规则很好玩，按这个规则，应该接受能求出值0的任何常量表达式的结果作为空指针。这个规则说明2-2和~-1都是空指针，然而用2-2或者~-1给指针赋值当然是个类型错误。这也不是我作为实现者所喜欢的规则。

11.2.4 overload关键字

在一开始，只有对一个名字明显地写出overload声明之后，C++才允许将它用到多于一

个命名之中（也就是说，“被重载”）。例如：

```
overload max;           // ``overload'' - obsolete in 2.0
int max(int,int);
double max(double,double);
```

我那时认为，如果不显式地声明重载意图，允许对两个函数采用同样名字会太危险。例如：

```
int abs(int);           // no ``overload abs''
double abs(double);    // used to be an error
```

当时对重载的担心有两个原因：

- 1) 担心可能出现无法检查的歧义性。
- 2) 担心除非程序员显式地说明哪些函数可能被重载，程序就可能无法正确地连接。

前一个担心已经被证明基本上是毫无根据的，在实际使用中发现的少量问题都通过顺序依赖性重载解析规则解决了。后一个担心则被证明，其基础就是C语言分别编译规则的一般性问题，与重载没有任何关系（见11.3节）。

而在另一方面，overload声明本身却变成了一个严重问题。它使人们无法合并对不同函数使用了同样名字的多段程序，除非在各个片段里都已经声明了有关名字的重载。在实际中常常都不是这种情况。典型的情况是，想要重载的名字可能就是C库函数的名字，在某个C头文件里声明。例如：

```
/* Header for C standard math library, math.h: */
double sqrt(double);
/* ... */

// header for C++ complex arithmetic library, complex.h:
overload sqrt;
complex sqrt(complex);
// ...
```

现在我们可以写：

```
#include <complex.h>
#include <math.h>
```

但是却不能写：

```
#include <math.h>
#include <complex.h>
```

因为只对sqrt的第二个声明使用overload是个错误。存在一些缓解这个问题的方法：重新安排声明，在头文件的使用上附加一些限制，将overload声明散布到所有的地方（只是为了）以防万一”。无论如何，我们发现这些伎俩都是不易处理的，除了可以对付一些最简单的情况。废除overload声明，彻底摆脱overload关键字，事情就好得多。

11.3 类型安全的连接

C连接极其简单，完全没有安全性。你声明了一个函数：

```
extern void f(char);
```

连接程序将轻率地将f连接到它的世界里的任一个f上。被连接的f可能是一个具有完全不同参数情况的函数，甚至根本不是函数。这通常会在运行时导致某种错误（核心卸载、段侵害等等）。连接问题有时会变得特别严重，因为它们的增长不是与程序的规模或者使用库的量成比例的。C程序员已经学会了如何在这个问题下生存。但无论如何，对于重载机制的需求使这个问题变得更急迫了。对于C++连接问题的任何解决方案又都必须继续保证能够调用C函数，也不增加任何复杂性或者开销。

11.3.1 重载和连接

在2.0之前的实现中有关C/C++对接问题的解决方法是，只要可能，就让C++函数产生的名字与假如用C来做产生的名字完全相同。这样，在那些C语言输出时不修改名字的系统中，C++对open()产生的名字也依然是open；而在那些C在名字前加下划线前缀的系统中，C++也产生_open；如此等等。

采用这种简单模式显然不足以对付重载函数。引进关键字overload，部分原因就是想利用它把复杂情况与简单情况区分开（另见3.6节）。

原来的解决办法与后面的方案类似，其基本思想就是将类型信息编码传递到连接程序的名字里（3.3.3节）。为了允许与C函数的对接，只对重载函数的第二个和其后版本的名字进行编码。这样，程序员可以写：

```
overload sqrt;
double sqrt(double);    // a linker sees: sqrt
complex sqrt(complex); // a linker sees: sqrt_F7complex
```

C++的编译程序将产生引用sqrt和sqrt_F7complex的代码。幸运的是，我只是在C++手册页的BUGS一节里写明了这种诡计。

2.0之前，在C++所用的重载模式与传统C连接模式对接时，会显示出了这两种方式的最坏的方面。我们必须解决三个问题：

- 1) 连接程序里类型检查的缺位。
- 2) overload运算符的使用。
- 3) C++与C程序段的连接。

对问题1的一种解决办法就是把函数的类型信息编码到每个函数的名字里。对问题2的一个解决办法是废除overload关键字。对问题3的解决办法是要求C++程序员明确说明一个函数需要使用C风格的连接。随之[Stroustrup, 1988a]：

“问题是，能否有一种基于这三个允诺的解决方案，它能够在不需要指明重载，在对C++程序员只造成最小不方便的条件下就能够实现。理想的解决方案应该是：

- 不需要修改C++语言。
- 提供类型安全的连接。
- 允许简单方便地与C连接。
- 不打破现有的C++代码。
- 允许使用(ANSI风格的)C头文件。
- 提供好的错误检查和错误报告。
- 成为构造库的好工具。

- 不增加运行时的开销。
- 不增加编译时的开销。
- 不增加连接时的开销。

我们还没能创造出这样的一种模式，使它能够严格地满足所有这些准则，但所采纳的模式是很接近的。”

很清楚，有关的解决方案需要对所有连接做类型检查。现在的问题变成如何去做，而又不必为每个系统重写新的连接程序。

11.3.2 C++连接的一种实现

首先需要对每个C++函数做编码，在后面附加它的参数类型。这样就能保证，如果一个程序能完成连接，那就要求每个函数调用必须有一个函数定义，而且在声明中所描述的参数类型与函数定义里描述的类型相同。例如，有了：

```
f(int i) { /* ... */ }           // defines f_Fi
f(int i, char* j) { /* ... */ }  // defines f_FiPc
```

下面的例子都能正确地处理：

```
extern f(int);                  // refers to f_Fi
extern f(int,char*);           // refers to f_FiPc
extern f(double,double);       // refers to f_Fdd

void g()
{
    f(1);                      // links to f_Fi
    f(1,"asdf");               // links to f_FiPc
    f(1,1);                    // tries to link to f_Fdd
                                // link-time error: no f_Fdd defined
}
```

剩下的问题是怎样去调用C函数，还有如何把C++函数“化装成”C函数。为了做到这些，程序员必须说明有关函数具有C连接。否则，它们就会被假定是C++函数，其名字将被编码。为表达函数具有C连接的情况，C++里引进了一个连接描述扩充：

```
extern "C" {
    double sqrt(double); // sqrt(double) has C linkage
}
```

这种连接描述并不影响使用sqrt()的程序的语义，而只是告诉编译程序，在目标代码里，对sqrt()的使用应该采用C的命名习惯。这意味着这个sqrt()的连接名将是sqrt或者_sqrt，或者其他什么在给定系统上C连接规则所要求的东西。也可以设想在某个系统上，C的连接规则就是如上所述的类型安全的C++连接规则，因此C函数sqrt()的连接名也就是sqrt_Fd。

自然，加上类型编码后缀只是实现技术的一个例子。它正是我们成功地用于Cfront的技术，而后又被其他人广泛地拷贝。这种技术的最重要性质就是简单，而且能与现存的连接程序一起工作。类型安全连接理想的这个实现并不是100%安全的，但在那个时候，100%安全的有用的系统极少，这种情况也具有普遍性。对Cfront中使用的编码模式（“名字压延”）的完整描述可以在[ARM, 7.2c节]找到。

11.3.3 回顾

我认为，对类型安全连接的要求提供合理的实现方法，并提供一种与其他语言连接的显式的旁路方式，这些东西的组合是一条正确的路。正如我们所期望的，新连接系统解决了问题，而又没有给用户强加某种他们无法接受的负担。此外，在将原有的C和C++代码转到新风格下的过程中，挖掘出了数目令人吃惊的连接错误。我在那时观察到的情况是：“转移到类型安全的连接，感觉就像是对C程序第一次运行lint——真有点害臊。”lint是一个很流行的工具，用于对C程序分别编译的单元做类型一致性检查 [Kernighan, 1984]。在开始引进的阶段，我试着去保存试验的踪迹。类型安全的连接在我们编译和连接的每个重要的C和C++程序里都查出了迄今尚未发现的错误。

还有一个令人吃惊之处，一些程序员已染上提供错误函数声明的恶习，他们常常简单地封住编译程序的嘴。例如，如果f()没有声明，调用f(1, a)将导致一个错误。在这个情况发生时，我一直朴素地期望程序员或者是为函数加上正确的声明，或者加上一个包含该声明的头文件。现在弄清楚了，还存在着第三种方式——直接提供某种符合这个调用的声明：

```
void g()
{
    void f(int ...); // added to suppress error message
    // ...
    f(1, a);
}
```

类型安全的连接将检查出这种懒散行为，只要一个声明与对应的定义不匹配，它就会报告出一个错误。

我们也发现了一个兼容性问题：人们常常是直接声明库函数，而不是去包含正确的头文件。我设想这样做的用意是想尽量减少编译时间，但产生的实际影响是，如果把有关代码移植到另一个系统上，这些声明有可能就是错误的。类型安全的连接也帮助我们捕捉到不多的几个这类的移植性问题（主要是在UNIX系统V和BSD UNIX之间）。

在实际地将一个类型安全的连接模式加到语言上之前，我们考虑了若干种不同的选择 [Stroustrup, 1988]：

- 不提供旁路，依靠工具处理C连接。
- 只对显式地用overload标记的函数提供类型安全的连接和重载。
- 只对不可能是C函数的那些函数（因为它们具有不能在C里表述的类型）提供类型安全的连接。

使用所采纳模式的经验使我确信，我对采用其他方式所假设的问题都是真实的。特别是在默认情况下把检查扩充到所有函数已经成为一种恩惠，混合使用C/C++已经变得很普及，任何使C/C++连接复杂化的东西都一定会造成许多伤害。

在用户那里引起的两类抱怨的细节直到现在还受到关注。对其中一种情况，我认为我们确实是做出了正确的选择。而对另一类我就不那么肯定了。

对声明为具有C连接的函数仍然采用C++的调用语义。这就是说，形式参数必须声明，实际参数必须在C++的匹配和歧义性控制规则的意义下能够与之匹配。有些用户希望具有C连接的函数遵从C的调用规则。采纳这种东西将允许更直接地使用C的头文件，但它也将使懒散的程序员能复辟C语言更弱的类型检查。反对为C引进特定规则的另一个论点是，为了使与

Pascal、Fortran、PL/I的连接能够更完整，程序员也要求能够支持从这些语言的函数里进行调用。例如，具有Pascal连接的函数应该隐式地将C风格的字符串转换到Pascal风格的字符串；对采用Fortran连接的函数提供引用调用，增加数组类型信息等。如果我们为C语言提供了特殊的服务，我们就将有义务给C++编译程序添加针对没有限制的一集语言的知识。顶住这种压力是正确的，虽然某种有效添加的服务确实能施惠于个别的混合程序设计语言系统的用户。在（仅有）C++语义的情况下，人们已经发现引用（3.7节）对于提供接口，与那些支持引用传递参数的语言连接方面很有用，例如与Pascal或者Fortran等连接。

另一方面，仅仅关注连接也带来了一个问题。这个解决方案并没有直接考虑在一个环境里支持混合语言程序设计和指向具有不同调用规则的函数指针的问题。利用C++的连接规则，我们能够直接描述遵循C++或者C调用规范的函数，但却无法直接去描述，某函数本身遵循C++规范，而它的参数却遵循C规范。有一个解决方法是间接地描述这件事情 [ARM, 118页]。例如：

```
typedef void (*PV)(void*,void*);  
  
void* sort1(void*, unsigned, PV);  
extern "C" void* sort2(void*, unsigned, PV);
```

在这里sort1()具有C++连接，它以一个到具有C++连接的函数的指针作为参数；而sort2()具有C连接，以一个到具有C++连接的函数的指针作为参数。这些都是很能说明情况的例子。在另一方面，考虑：

```
extern "C" typedef void (*CPV)(void*,void*);  
  
void* sort3(void*, unsigned, CPV);  
extern "C" void* sort4(void*, unsigned, CPV);
```

这里sort3()具有C++连接，以一个到具有C连接的函数的指针作为参数；sort4()具有C连接，以一个到具有C++连接的函数的指针作为参数。这些把语言能够描述什么的限制向前推进了，但是也很难看。看起来其他方式也不大会更有吸引力：你可以将调用规则引进类型系统，或者广泛使用调用标签来处理这类混合的调用规则。

连接，跨语言调用，跨语言对象传递，这些东西在本质上就是非常困难的问题，也有许多依赖于实现的方面。这也是一个变化的领域，其基础规则也会随着新语言、硬件体系结构以及实现技术的发展而不断改变。我预计我们还没有听到这个事情的终结。

11.4 对象的建立和复制

在这些年里，我一直常规地接到请求，要求能有禁止各种操作的语言特征（在这十年里大约每个星期有2~3次）。提出的理由千变万化。有些人希望能优化类的实现，所采用的方式只能在对该类的对象绝不执行复制、派生或堆栈操作的情况下才能完成。在另一些情况中，例如那些想要表示真实世界对象的对象，所要求的语义就根本不希望包括C++以默认方式提供的所有东西。

对于这类要求的大部分回答都是在2.0的工作期间发现的：如果你希望禁止某些东西，就应该把完成它的操作定义为一个私用的成员函数（2.10节）。

11.4.1 对复制的控制

为了禁止对类X对象的复制，只要简单地把复制建构函数和赋值都定义为私用的：

```
class X {
    X& operator=(const X&); // assignment
    X(const X&); // copy constructor
    // ...
public:
    X(int);
    // ...
};

void f()
{
    X a(1); // fine: can create Xs
    X b = a; // error: X::X(const X&) private
    b = a; // error: X::operator=(const X&) private
}
```

当然，类X的实现者还是可以复制X对象，不过在实际情况中这是可以接受的，甚至根本就是所需要的。不幸的是，我已经记不起来是谁最先想到了这个办法，我怀疑可能就是我自己，参见[Stroustrup, 1986, 172页]。

我个人认为，将复制操作定义为默认的也是一种不幸。我对自己的许多类都禁止了复制操作。但无论如何，C++是从C那里继承来的默认赋值和复制建构函数，它们也确实经常是需要的东西。

11.4.2 对分配的控制

另外一些很有用的作用也可以通过将操作声明为私用而获得。例如，只要将析构函数声明为私用就可以避免堆栈和全局分配。这样做还能防止随便使用delete：

```
class On_free_store {
    ~On_free_store(); // private destructor
    // ...
public:
    static void free(On_free_store* p) { delete p; }
    // ...
};

On_free_store glob1; // error: private destructor

void f()
{
    On_free_store loc; // error: private destructor
    On_free_store* p = new On_free_store; // fine
    // ...
    delete p; // error: private destructor
    On_free_store::free(p); // fine
}
```

当然，这种类的典型使用需要一个高度优化的自由存储分配系统，或者其他能够取得自由存储上对象的优点的语义。

与此相反的作用——只允许全局和堆栈变量，禁止自由空间分配——也可以做到，只要声明一个不大寻常的operator new()：

```
class No_free_store {
    class Dummy { };
    void* operator new(size_t, Dummy);
    // ...
};

No_free_store glob2; // fine

void g()
{
    No_free_store loc; // fine
    No_free_store* p = new No_free_store; // error:
        // no No_free_store::operator new(size_t)
}
```

11.4.3 对派生的控制

私用的析构函数也能制止派生。例如：

```
class D : public On_free_store {
    // ...
};

D d; // error: cannot call private base class destructor
```

这实际上说明，带有私用析构函数的类与抽象类在逻辑上互为补集。从On_free_store类不可能做派生，所以，对On_free_store类的虚函数调用就不需要虚函数机制。但是无论如何，我还没有看到当前的任何编译系统能够基于这种情况的做一些优化。

后来Andrew Koenig发现，甚至不需要对能做哪些分配强加上任何限制，同样也可以防止派生：

```
class Usable_lock {
    friend Usable;
private:
    Usable_lock() {}
};

class Usable : public virtual Usable_lock {
    // ...
public:
    Usable();
    Usable(char*);
    // ...
};

Usable a;

class DD : public Usable { };
```

```
DD dd; // error: DD::DD() cannot access
        // Usable_lock::Usable_lock(): private member
```

这种做法依赖于有关的规则：一个派生类必须（隐含地或显式地）调用其基类的建构函数。

这种例子通常更多的是一种智力嗜好，而不是什么具有重要实际意义的技术。这可能也是为什么对于它们的讨论如此广泛的根本原因。

11.4.4 按成员复制

开始时，默认定义的初始化和赋值都采用按位进行复制的方式。如果某个带有赋值的类的对象被当作另一个没有带赋值的成员使用时，这种做法就会带来一些问题：

```
class X { /* ... */ X& operator=(const X&); }

struct Y { X a; };

void f(Y y1, Y y2)
{
    y1 = y2;
}
```

在这里，`y2.a`将以按位的方式复制到`y1.a`。这肯定是错的。而之所以造成这种结果的原因很简单，就是由于忽视了实际存在的赋值和复制建构函数。经过一些讨论，也是在Andrew Koenig的催促下，我们采纳了最明显的解决方案：对象的复制被定义为对所有非静态的成员和基类成员的按成员复制。

这个定义实际上断言说，`x=y` 的意义就是`x.operator=(y)`。这里有一个非常有意思的隐喻。考虑：

```
class X { /* ... */ };
class Y : public X { /* ... */ };

void g(X x, Y y)
{
    x = y; // x.operator=(y): fine
    y = x; // y.operator=(x): error x is not a Y
}
```

按照默认方式，对`x`的赋值就是`X& X::operator=(const X&)`，所以`x=y`是合法的，因为`Y`是由`X`经过公有派生得到的类。这通常被称作切片，因为`y`的一个切片被赋给了`x`。复制建构函数也按类似的方式处理。

我从实践的观点考虑，总对切片抱有戒心，但又看不到任何阻止它的方法，除非是加上一条非常特殊的规则。还好，在那个时候我又得到了一个来自Ravi Sethi的恰好是关于“切片语义”的独立请求，他从理论和教育的观点出发希望有这种语义：除非你能将派生类的对象赋值给其公有基类的对象，否则这就会是C++里面仅存的一个地方，在这里派生类的对象不能被用在基类对象可以使用的地方。

上面这些还给默认赋值操作留下了一个问题：指针成员是复制了，但是它们的所指却没有复制。这种情况几乎总是错误的，而由于与C语言的兼容性，又不能禁止。当然，编译系统很容易对这种情况提出一个警告，只要一个有指针成员的类采用了默认的复制建构函数或者

赋值做复制。例如：

```
class String {
    char* p;
    int sz;
public:
    // no copy defined here (sloppy)
};

void f(const String& s)
{
    String s2 = s; // warning: pointer copied
    s2 = s;        // warning: pointer copied
}
```

C++里按照默认方式的赋值和复制建构函数有时被称为浅复制，也就是说，它只复制类的成员，但不复制由某些成员所指的对象。另一种方式就是递归地复制被指对象（这常被称作深复制），在这里只能显式定义。由于有自引用对象的可能性，对于这件事也很难有其他办法。一般地说，试图定义完成深复制的赋值操作并不是聪明的办法；而定义一个（虚的）拷贝函数通常都是更好的选择（见 [2nd, 217~220页] 和 13.7 节）。

11.5 记法约定

我的目标是允许用户描述每个运算符的意义，只要这样做有价值，只要它不会严重地干扰预先定义的语义。如果我原来就毫无例外地允许重载所有的运算符，或者不允许重载任何对于类对象已经有了预定义意义的运算符，事情都会简单得多了。结果只能是一种折衷，每个人都不是很满意。

几乎所有讨论和大部分问题都出自于那些不符合二元算术运算符或前缀算术运算符的常规模式的运算符。

11.5.1 灵巧指针

在2.0之前，指针间接运算符`->`是不能由用户重新定义的。这就使人很难建立起某种对象的类，这些对象按设想能够以某种“灵巧指针”的方式进行活动。这里的原因也很简单，在定义运算符重载时，我把`->`看成是一个对其右边运算对象（成员名）采用某种非常特殊的规则的运算符。我记得在俄勒冈Mentor Graphics的一次会议上，Jim Howard跳起来，绕过一个很大的会议桌走到黑板前，纠正了我的这个错误观点。他指出，运算符`->`可以被看作一个一元运算符，其结果再应用到成员的名字上。后来我继续在重载机制上工作时就采用了这个看法。

这实际上是说明，如果要使用某个`operator->()`函数的返回类型，它必须是一个指针，指向某个类或者某个类的一个对象，`operator->()`的定义就是用于它们的。例如：

```
struct Y { int m; };

class Ptr {
    Y* p;
    // ...
public:
```

```

    Ptr(Symbolic_ref);
    ~Ptr();

    Y* operator->()
    {
        // check p
        return p;
    }
};

```

在这里，`Ptr`的定义使每个`Ptr`的行为就像是指向`Y`类对象的指针，除了在每次访问时还要执行一些适当的操作之外。

```

void f(Ptr x, Ptr& xr, Ptr* xp)
{
    x->m;    // x.operator->() ->m; that is, x.p->m
    xr->m;   // xr.operator->() ->m; that is, xr.p->m
    xp->m;   // error: Ptr does not have a member m
}

```

如果将这种类定义为模板（15.9.1节）就会很有用处 [2nd]：

```

template<class Y> class Ptr { /* ... */ };

void f(Ptr<complex> pc, Ptr<Shape> ps) { /* ... */ }

```

当`->`的重载在1986年第一次实现时，这个问题就已经被认识到了。不幸的是，那还是在模板机制能够使用的许多年之前。只有有了模板，才能实际地写出这样的代码。

对于普通指针，`->`只是作为一元的`*`和`[]`的某些使用的同义词。例如，对一个`Y* p`，下面的关系成立：

```
p->m == (*p).m == p[0].m
```

与平常一样，对于用户定义运算符不可能提供这种保证。当然，如果需要的话，就可以提供这种等价关系：

```

class Ptr {
    Y* p;
public:
    Y* operator->() { return p; }
    Y& operator*() { return *p; }
    Y& operator[](int i) { return p[i]; }
    // ...
};

```

运算符`->`的重载对很大一类有意义的程序是非常重要的，这不是一个次要的只是引起好奇心的事物。因为间接是一个非常重要的概念，对`->`的重载为在程序里表达这个概念提供了一种清晰、直接而有效的方式。对`->`运算符还有另一种看法，那就是将它看作C++提供的一种受限的、但也是非常有用的委托机制的形式（12.7节）。

11.5.2 灵巧引用

当我决定允许对运算符`->`的重载时，自然地就考虑到是否也能够对运算符`.`做类似的

重载。

那时我认为如下的说法是具有结论性的：如果obj是一个类对象，那么这个对象所属的类里对每个成员m，obj.m就已经有了一种意义。我们不想通过重新定义某些内部运算使语言成为可变的（虽然这个规则已经被违反了，例如那些超出了真正必要的对于=和一元&的重新定义）。

如果我们允许对类x里的.进行重载，我们将可能无法通过正常途径访问x的成员；有可能不得不去用一个指针和->，但是有可能->和&也已经被重新定义了。我希望要的是一个可扩充的语言，而不是一个变化无常的东西。

这些论据都很有分量，但却也不是结论性的。特别是，在1990年Jim Adcock提出了允许对运算符.进行重载的建议，以与运算符->完全一样的方式。

为什么人们希望能重载operator.()呢？实际上就是为了提供一个类，使它的行为就像是另一个实际完成工作的类的“句柄”或者“代理”。作为例子，在这里有一个用在有关operator.()重载问题的早期讨论中的多精度整数类：

```
class Num {
    // ...
public:
    Num& operator=(const Num&);
    int operator[](int);           // extract digit
    Num operator+(const Num&);
    void truncateNdigits(int);   // truncate
    // ...
};
```

我很希望有一个类RefNum，其行为就像是Num&，但还可以执行另外一些操作。例如，如果我能写：

```
void f(Num a, Num b, Num c, int i)
{
    // ...
    c = a+b;
    int digit = c[i];
    c.truncateNdigits(i);
    // ...
}
```

而且我还会希望能写：

```
void g(RefNum a, RefNum b, RefNum c, int i)
{
    // ...
    c = a+b;
    int digits = c[i];
    c.truncateNdigits(i);
    // ...
}
```

条件是operator.()已经按恰好与operator->()平行的方式定义好。我们首先试了下面这种明显的RefNum定义方式：

```
class RefNum {
    Num* p;
```

```

public:
    RefNum(Num& a) { p = &a; }
    Num& operator.() { do_something(p); return *p; }
    void bind(Num& q) { p = &q; }
};

```

不幸的是这并不能产生正确的效果，因为在所有情况下都不会显式地提到运算符：

```

c = a+b;           // no dot
int digits = c[i]; // no dot
c.truncateNdigits(i); // call operator.()

```

为此我们将必须写出一些前向函数，以保证当运算符作用到RefNum上的时候，能够执行正确的动作：

```

class RefNum {
    Num* p;
public:
    RefNum(Num& a) { p = &a; }
    Num& operator.() { do_something(p); return *p; }
    void bind(Num& q) { p = &q; }

    // forwarding functions:

    RefNum& operator=(const RefNum& a)
    { do_something(p); *p = *a.p; return *this; }
    int operator[](int i)
    { do_something(p); return (*p)[i]; }
    RefNum operator+(const RefNum& a)
    { do_something(p); return RefNum(*p + *a.p); }
};

```

这明显是非常令人讨厌的。由此，包括Andrew Koenig和我在内的许多人开始考虑将operator.()应用到RefNum的每个运算上的作用。按照这种方式，RefNum原来的定义就能使原来例子像所希望的那样工作了（这也是开始时的期望）。

但无论如何，以这种方式使用operator.()隐含着一个问题：要访问RefNum的成员，你就必须用一个指针：

```

void h(RefNum r, Num& x)
{
    r.bind(x); // error: no Num::bind
    (&r)->bind(x); // ok: call RefNum::bind
}

```

看起来，在什么是operator.()的最好解释的问题上C++社团出现了分裂。我倾向于这样的观点，如果应该允许operator.()，那么它就应该既能显式调用也能隐式调用。毕竟定义operator.()的原因是为了避免写前向函数。除非隐式的.也同样用operator.()进行解释，否则我们就还需要写大量的前向函数，或者我们就应该戒绝运算符重载。

如果我们能定义operator.()，那么a.m和(&a)->m的等价性将不会按定义自动继续存在了。当然，可以通过重新定义operator&()和operator->()，使它们能够与operator.()相匹配，因此我个人并不把这看成是一个重要问题。当然，如果我们真的这样做了，而后就再没有办法去访问有着灵巧引用的类的成员了。例如RefNum::bind()将会变

成完全不可访问的东西。

这个问题重要吗？一些人的回答是“不，就像常规引用一样，灵巧引用也不应该能重新约束到新的对象上。”但按照我的经验，灵巧引用常常需要某种重新约束操作或者某些其他操作，以使它真正能有用处。看起来大部分人同意这个认识。

现在我们就陷入了一个窘境，或者是保持`a.m`和`(&a)->m`的等价性，或者是保持对有灵巧引用的类成员的访问权，但不可能两者兼得。

脱离两难境地的一种方法是，对于`a.m`来说，只有在引用类本身并没有一个名为`m`的类成员时才去使用`operator.()`，这恰好是我所喜爱的解决方法。

但不管怎样，对于重载`operator.()`的重要性还不存在一个共识，因此`operator.()`还不是C++的一部分，争论还在激烈进行之中。

11.5.3 增量和减量的重载

增量运算符`++`和减量运算符`--`都是用户可以定义的运算符。但是，Release 1.0并没有提供区分前缀或后缀应用的机制。有了：

```
class Ptr {
    // ...
    void operator++();
};
```

在两种情况中使用的都将是同一个`Ptr::operator++()`：

```
void f(Ptr& p)
{
    p++; // p.operator++()
    ++p; // p.operator++()
}
```

有些人，特别是Brian Kernighan指出，从C语言的观点看，这种限制是很不自然的，它也阻止了人们去定义那种能够用来取代常规指针的类。

在设计C++的运算符重载机制时，我当然已经考虑过区分前缀和后缀增量的问题，但我当时的决定，为了表示这种区分而增加语法形式似乎不太值得。后来几年里，收到的有关建议的数目使我确信原来的决定并不正确，也推动我去寻找某种能分别表达前缀/后缀的最小修改。

我考虑了最明显的解决办法，在C++里增加关键字`prefix`和`postfix`：

```
class Ptr_to_X {
    // ...
    X& operator prefix++(); // prefix ++
    X operator postfix++(); // postfix ++
};
```

或者：

```
class Ptr_to_X {
    // ...
    X& prefix operator++(); // prefix ++
    X postfix operator++(); // postfix ++
};
```

但是我又从最不喜欢增加关键字的人们那里听到了熟悉的愤怒吼声。人们建议了几个并不涉及新关键字的方案。例如：

```
class Ptr_to_X {
    // ...
    X& ++operator(); // prefix ++
    X operator++(); // postfix ++
};
```

或者

```
class Ptr_to_X {
    // ...
    X& operator++(); // prefix because it
                      // returns a reference
    X operator++(); // postfix because it
                      // doesn't return a reference
};
```

我觉得前一个太做作，而后一个又太微妙。最后我停止在：

```
class Ptr_to_X {
    // ...
    X& operator++(); // prefix: no argument
    X operator++(int); // postfix: because of
                      // the argument
};
```

这个可能是既做作而又微妙，但是它能用，又不需要新的语法，也有一个逻辑上的说法去对付那些愤怒的人。其他一元运算符都是前缀的，在定义为成员函数时都没有参数。这里“古怪”而无用的空int参数专用于指明这是一个古怪的后缀运算符。换句话说，在后缀形式中++是放在第一个（真实的）运算对象和第二个（空的）参数之间，并因此是后缀的。

这些解释确实很需要，因为这个机制是独特的，因此就有些讨厌。如果让我选择，我可能还是会引进prefix和postfix关键字，但在那时这样做是明显不可行的。无论如何，真正最重要的一点是这个机制能行，也能够被理解，但实际上只有少数程序员使用过它。

11.5.4 重载->*

运算符->*也作为可以重载的，根本原因是没有任何理由说它为什么不行（由于正交性，如果你非要有一个理由的话）。转而又发现，这一点在表达约束操作方面很有用，那些东西在语义上总得设法与内部的->*（13.11节）平行。在这里不需要特殊规则，->*的行为就像其他任何二元运算符。

运算符.*没有被包括到程序员可以重载的运算符之中，个中理由与运算符.完全一样（11.5.2节）。

11.5.5 重载逗号运算符

在Margaret Ellis的推动下，我允许了对逗号运算符的重载。简单地说，我在那时没有发现不能这样做的理由。实际上确实有一个理由：a,b对所有a和b都已经有定义了，允许重载就使程序员能够改变内部运算符的意义。还好，只有当a和b都是类对象时他们才能这么做。

也出现了若干operator, () 的实际应用。接受它基本上就是为了一般性。

11.6 给C++增加运算符

永远不可能有足够的运算符能满足每个人的口味。实际上，看起来除了极少数人从根本上反对一切运算符之外，每个人都想有几个额外的运算符。

11.6.1 指数运算符

为什么C++不能有一个指数运算符？本来的原因就是C语言里没有。C运算符的语义被假定说应该足够简单，以使它们中的每一个在典型的计算机上都能对应于一条机器指令。指数运算符不能满足这个准则。

为什么在我第一次设计C++时不立即增加一个指数运算符呢？我的目标是提供抽象机制而不是新的基本运算符。加一个指数运算符必然要对内部类型的算术给出一种意义，这里是C语言的领地，我早已决定避免去改变它。进一步说，C语言以及而后的C++都常常因为有太多的运算符，还带有各种各样的混乱的优先级而受到批评。即使在这些威慑之下，我还是考虑过增加一个指数运算符，如果不是存在太多技术问题的话也可能就这样做了。我并不能完全确定在一个有重载和在线函数的语言里还真正需要一个指数运算符，但是简单地加上运算符，以平息反复出现的要求它的论断也还是很诱人的。

人们希望的指数运算符是`**`，这将导致一个问题，因为`a ** b`可以是一个合法的C语言表达式，其中涉及到对一个指针的间接操作：

```
double f(double a, double* b)
{
    return a**b; // meaning a*(b)
}
```

此外，看来在指数运算符的各个方面也还有许多不同意见，例如这个运算符应该具有的优先级：

```
a = b**c**d; // (b**c)**d or b** (c**d) ?
a = -b**c;   // (-b)**c or -(b**c) ?
```

最后，我也极不情愿去描述指数运算的数学性质。

在当时，这些理由都使我认为，要更好地为用户服务，可能还是应该去关注其他论题。回过头看，这些问题实际上都是可以克服的，真实的问题是“这样做值不值得”？当1992年Matt Austern向C++标准化委员会提出了一个完整的建议书时（第6章），这个问题也走到了头。在通往委员会的路上这个建议已经收到了许多评注，并成为网络上许多争论的根源。

为什么人们希望一个指数运算符？

- 他们在Fortran里已经习惯于用它。
- 他们相信一个指数运算符可能比一个指数函数优化得更好。
- 在那些实际上由物理学家和指数运算符的其他基本用户写出的表达式里，用一个函数调用确实很难看。

这些理由足以推翻技术问题和反对意见吗？另外，有关的技术问题应该如何解决？扩充工作组讨论了这些问题，并最终决定不加入指数运算符。Dag Brück总结了有关理由：

- 运算符是为了提供记法上的方便性，而不是为了提供新功能。工作组的成员代表了科学/工程计算的很大一部分用户，他们指出，这个运算符的语法只能提供次要的语法方便。
- C++的每个用户都必须学习这个新特征。
- 用户已经强调了用他们自己特殊的指数函数取代系统中默认函数的重要性，如果用一个内部的运算符，这件事就不可能了。
- 这个建议的诱因还是不够充分。特别是仅通过查看一个30 000行的Fortran程序，不能就做出结论说该运算符将在C++里广泛使用。
- 这个建议要求增加一个新运算符并增加另一个优先级，这就增加了语言的复杂性。

这个陈述多少有些有意识地少说了讨论的深度。例如，有的委员会成员审阅了大量的公司代码，查看其中指数运算符的使用情况，没有发现它的使用像一些人所陈述的那么具有关键性。另一个关键性认识是，在被检查的Fortran代码里， ** 的主要出现形式是 a^{**n} ，在这里n是很小的整数字面量；在大部分情况中写 a^*a 和 a^*a^*a 的方式是可行的。

从长远的观点看，接受这个建议是不是会更省事？这个问题还要继续看。无论如何，还是让我来提出一些技术性问题。哪个运算符作为C++的指数运算符最好？C语言使用了ASCII字符集里除了@和\$之外的所有图形字符，而由于一些原因，这两个符号都不合适。运算符!、~、*~、^^甚至单独的^（在某一个运算对象不是整数的情况下）都被考虑过。由于@、\$、~、!是本国字符，并不出现在所有键盘上（见6.5.3节）；许多人认为@和\$用在这里太难看。单词^和^^被C程序员读成“不相交或”。另一个附加限制是，应该能按其他算术运算符同样的方式，将指数运算符与赋值运算符组合，例如+和=组合成+=。这就又删掉了!=，因为!=已经有了自己的意思。Matt Austern最后选定*^，这可能是最好的选择了。

所有其他的技术问题都可以按照它们在Fortran里的方式解决。这也是最明智的解决方案，可以节省大量工作。Fortran在它的领域里就是标准，要偏离一个事实上的标准需要有非常明显的原因。

在这里，让我再重新考虑以 ** 作为C++的指数运算符。无论如何我已经说明了，如果采用传统技术，这样做是行不通的，但在重新检查这个问题时我认识到，C的兼容性问题是可以通过某些编译技巧克服的。假定我们引进了 ** 运算符，我们实际上还是能处理这里的不兼容性，当第二个运算对象是指针时将它的意义定义为“间接和乘”：

```
void f(double a, double b, int* p)
{
    a^{**}b; // meaning pow(a,b)
    a^{**}p; // meaning a^{*}(*p)
    ^*a; // error: a is not a pointer
    ^*p; // error: means ^(*p) and *p is not a pointer
}
```

为了适应这个语言， ** 当然应该是一个单词。这也就意味着，在 ** 出现为一个声明符时它必须被解释为双重的间接：

```
char^{**} p; // means char ^ * p;
```

最主要问题是，为了使 $a/b^{**}c$ 能够表达数学里的意思，即 $a/(b^{**}c)$ ， ** 的优先级必须高于*。而另一方面， $a/b^{**}p$ 在C语言里的意思是 $(a/b)^{*}(*p)$ ，这样它就会不动声色地改变

意义，变成了 $a / (b * (*p))$ 。我预计这种代码在C和C++里是极其罕见的，如果我们真的决定再提供一个指数运算符，那么打破这样的代码也是值得的——特别是为了让编译程序提出警告，说明这里可能出现意义改变是非常简单的事。当然，我们现在决定了不增加指数运算符，这些讨论就完全是学术上的了。我很喜欢看到我这种半严肃地建议使用 $**$ 所引起的恐怖。由例如指数应该被表达为 $pow(a, b)$ 、 $a^{**}b$ 或者 $a^{*^}b$ 这样一个并不重要的语法问题，居然产生出这么大的热量，我一直感到好玩和不解。

11.6.2 用户定义运算符

我能不能通过设计出一种使用户可以定义自己的运算符的机制，以避免关于指数运算符的整个讨论呢？这样也能一般性地解决缺乏运算符的问题。

当你真的需要运算符时，你将不可避免地发现由C和C++提供的运算符集合对于表示所需要的运算符总是不够的。解决的办法当然是去定义函数。但无论如何，如果你能够对某些类说

$a^{*}b$

的时候，函数调用形式如

```
pow(a, b)
abs(a)
```

看起来就不那么令人满意了。随之，人们就要求能为下面这些形式定义意义：

```
a pow b
abs a
```

这是可以做到的，Algol 68展示了一种做法。进而，人们又可能要求能够为下面这些形式定义意义：

```
a ** b
a // b
| a
```

如此等等。这个也能做到。真正的问题是，允许用户定义运算符本身是不是一件值得做的事情。我研究了这件事情 [ARM]：

“这个扩充，无论如何，都必然隐含着语法分析复杂性的显著提高，以及在可读性方面不大好确定的收获。它还要求或者是允许用户为新运算符确定约束强度和结合方式，或者是为所有的用户定义运算符固定这些属性。在任何一种情况下，例如下面这样的表达式的约束关系

```
a = b**c**d; // (b**c)**d or b** (c**d) ?
```

都会令许多用户感到吃惊和烦恼。我们还必须解决它们与常规运算符在语法方面的冲突。考虑下面的情况，假定 $**$ 和 $//$ 都被定义成二元运算符了：

```
a = a**p; // a**p OR a*(p)
a = a//p;
*p = 7; // a = a*p = 7; maybe? "
```

作为必然的推论，用户定义的运算符或者必须限制到常规的字符上，或者要求带一个辨别前

缀, 例如. (圆点):

```
a pow b; // alternative 1
a .pow b; // alternative 2
a .** b; // alternative 3
```

对用户定义运算符也必须给定一个优先级。做这件事的最简单方式就是将一个用户定义运算符的优先级描述为与某个内部运算符一样。但无论如何, 对于“正确”定义指数运算符而言这种方式又是不够的。为此我们需要某些更精巧的东西, 例如:

```
operator pow: binary, precedence between * and unary
```

此外我还非常担心程序的可读性问题, 在有了带有用户定义优先级的用户定义运算符之后, 情况究竟会如何。例如, 在不同程序设计语言里对指数使用了不止一种优先级, 因此人们可能为pow定义不同的优先级。例如:

```
a = - b pow c * d;
```

在不同程序里就会有不同的分析结果。

另一种简单的选择是给所有用户定义运算符同样的优先级。这种选择初看起来很有吸引力, 但后来发现, 甚至我和我那时最接近的两个同事, Andrew Koenig和Jonathan Shaprio, 都无法在采用哪个优先级方面达成一致。最明显的候选是“很高”(例如恰好高于乘法)和“很低”(例如, 恰好高于赋值)。不幸的是, 一个人认为很理想而另一个却觉得太荒唐的情况层出不穷。看起来甚至对于最简单的例子, 要给出一个惟一“正确的”优先级也非常困难。看下面这个例子:

```
a = b * c pow d;
a = b product c pow d;
a put b + c;
```

这样, C++不支持用户定义运算符。

11.6.3 复合运算符

C++支持对一元和二元运算符的重载。我还想到, 支持复合运算符的重载也可能会很有用。在ARM里, 我用下面方式解释了这个想法:

“例如, 下面的两个乘法:

```
Matrix a, b, c, d;
// ...
a = b * c * d;
```

可以通过一个特别定义“双重乘法”运算符的方式实现, 例如采用如下形式:

```
Matrix operator * * (Matrix&, Matrix&, Matrix&);
```

这将使上面的语句能够解释为:

```
a = operator * * (b, c, d);
```

换句话说, 看到如下声明:

```
Matrix operator * * (Matrix&, Matrix&, Matrix&);
```

编译程序将寻找重复出现的Matrix相乘模式, 并调用这个函数作为它们的解释。与此不同的

或者太复杂的模式还继续用常规的（一元和二元）运算符处理。

这个扩充已经多次被独立地提出，作为在科学计算中处理用户定义类型的常见模式的有效方法。例如：

```
Matrix operator = * + (
    Matrix&,
    const Matrix&,
    double,
    const Matrix&
);
```

可以用于处理下面这样的语句：

```
a=b*1.7+d; "
```

很自然，在这种声明中空格放置的位置是非常重要的。换一种方式，也可以用其他符号来指明运算对象的位置：

```
Matrix operator.=.*.+.( 
    Matrix&,
    const Matrix&,
    double,
    const Matrix&
);
```

我在ARM之前还没有看到有其他发表了的文献里解释这种思想，但这是在代码生成器里常见的一种技术。我认为这种思想能够支持优化的向量和矩阵运算，但我一直没有时间去对它做充分开发，以确认这件事情。它也可能作为支持一种老技术的记法形式，以便去定义能够对几个参数执行某种复合运算的函数。

11.7 枚举

C语言的枚举是一个半生不熟的古怪概念。枚举并不是C语言原始概念的一部分，很明显是作为一种让步，非常勉强地引进语言中，提供给那些强调需要一种比C的无参宏更实质一些的符号常数形式的人们。因此，一个C枚举值的类型就是int，被声明为“枚举类型”变量的值也是如此。用int可以自由地给枚举变量赋值。例如：

```
enum Color { red, green, blue };

void f() /* C function */
{
    enum Color c = 2; /* ok */
    int i = c;         /* ok */
}
```

我在自己所希望支持的程序设计风格中并不需要枚举量，也没有特别的意愿去插手有关枚举的事情，所以C++就直接采纳了C的规则，没做任何改变。

不幸的是（或者说是幸运，如果你喜欢枚举），ANSI C委员会给我遗留下一个问题。它改变了，或者说是清理了枚举的定义，把指向不同枚举类型的指针看成是不同的类型：

```
enum Vehicle { car, horse_buggy, rocket };
```

```
void g(pc,pv) enum Color* pc; enum Vehicle* pv;
{
    pc = pv; /* probably illegal in ANSI C */
}
```

我们在这个问题上进行的讨论持续了极长的时间，涉及到一些C专家，如David Hanson、Brian Kernighan、Andrew Koenig、Doug McIlroy、David Prosser和Dennis Ritchie。这个讨论还没有完全成为结论性的——这本身就是一个坏兆头——但已经达成一个统一的意见：该标准的意图是指明上述例子违法，但也可能遗留下一个漏洞，如果Color和Vehicle被用同样数量的存储区表示时（这也是普遍的情况），它就有可能接受这个例子。

由于函数重载问题，我实在没办法接受这种不确定性。例如：

```
void f(Color*);
void f(Vehicle*);
```

必须或者是作为一个函数声明了两次，或者是被看作两个重载函数。我可不希望接受任何模棱两可的话，或者是依赖于实现。类似地，

```
void f(Color);
void f(Vehicle);
```

必须或者是声明的同一个函数，或者是两个重载的函数。在C语言和ARM之前的C++里，这些声明都作为一个函数声明了两次。当然，最清晰的方式是把每个枚举看成是一个独立的类型。例如：

```
void h() // C++
{
    Color c = 2; // error
    c = Color(2); // ok: 2 explicitly converted to Color
    int i = c; // ok: col implicitly converted to int
}
```

许多人早就提出，要求采纳这个解决方案，每次我与C++程序员讨论枚举时总会有人提。我怀疑我的行动还是太性急——虽然经过很多个月的拖延，以及向C和C++专家无穷无尽的咨询——但终归还是为将来获得了一个最好的结论。

11.7.1 基于枚举的重载

在把每个枚举声明作为一个单独的类型之后，我还忘记了一些非常明显的东西：一个枚举是用户定义的一个单独类型。因此它也就是一个用户定义类型，像一般的类一样。因此基于枚举就应该能做运算符的重载。Martin O'Riordan在一次ANSI/ISO会议上指出了这一点。他和Gag Brück一起做出了有关细节，基于枚举的重载已经被C++接受了。例如：

```
enum Season { winter, spring, summer, fall };

Season operator++(Season s)
{
    switch (s) {
        case winter: return spring;
        case spring: return summer;
        case summer: return fall;
        case fall: return winter;
    }
}
```

```

    }
}

```

我用开关避免整型算术运算和强制。

11.7.2 布尔类型

最常见一个枚举就是：

```
enum bool { false, true };
```

每个重要程序里都会有它或者它的兄弟姐妹：

```
#define bool char
#define Bool int
typedef unsigned int BOOL;
typedef enum { F, T } Boolean;
const true = 1;
#define TRUE 1
#define False (!True)
```

各种变化是无穷无尽的。更糟的是，许多变形还隐含着语义上的细微变化，许多变形在一起使用时还会与其他变形产生冲突。

当然，这个问题在许多年前就广为人知了。Dag Brück和Andrew Koenig觉得需要对此做点什么：

“关于C++里的布尔数据类型的想法是一个信仰问题。有些人，特别是来自Pascal或者Algol的人们，认为C语言里居然没有这样一个类型是很荒诞的，更别说是C++了。另一些人，特别是来自C的人们，认为任何人去操心向C++里加入这样一个类型那才真是荒唐呢。”

很自然，第一个想法就是定义一个枚举。但是，Dag Brück和Sean Corfield检查了数以十万行计的C++代码，发现布尔类型的大部分使用方式都需要在它与int之间自由地来回转换。这就意味着，如果真的定义一个布尔枚举，那就会打破太多的现存代码。那么为什么还要为布尔类型操心呢？

- 1) 布尔数据类型是生活中的一个事实，无论它是不是C++标准的一个部分。
- 2) 存在许多互相冲突的定义，将使方便安全地使用任何布尔类型变得非常困难。
- 3) 许多人希望有基于布尔类型的重载。

有点出乎我的意料，ANSI/ISO接受了这个论断，因此bool现在已经是C++里的一个特别的整类型了，带有字面量true和false。非零值可以隐式地转换到true，true可以隐式地转换到1。零值可以隐式地转换到false，而false可以隐式地转换到0。这样也就保证了高度的兼容性。

第12章 多重继承

因为你有
父亲和母亲 :-(
——comp.lang.c++

多重继承的时间表——普通基类——虚基类——支配规则——对象布局模型——从基类出发的强制——方法组合——有关多重继承的论战——委托——重命名——基和成员初始式

12.1 引言

多重继承就是允许有两个或者更多的基类的能力，在大部分人的心里，它应该是2.0的特征。那时我并不赞同，因为我觉得，从实践上看，对类型系统的整体改进是远远更为重要的事情。

还有，将多重继承加进Release 2.0也是一个错误。多重继承远不如参数化类型那么重要——而对于另一些人而言，参数化类型的重要性又不如异常处理。但在实际上，以模板形式出现的参数化类型到了Release 3.0才出现，异常则出现得更晚。我对参数化类型的渴望也远远超过了我对多重继承的渴求。

那时选择去做多重继承的工作有许多原因：设计能够进一步前进；将多重继承置入C++的类型系统，并不需要做多少扩充；有关的实现可以在Cfront里面完成。另一个因素则完全是不理性的：看起来没人怀疑我能够有效地实现模板机制，而对多重继承就是另一个情况，大家广泛认为很难有效地实现。例如Brad Cox在他关于Object C的书里面对C++的总结中就断言说，将多重继承加入C++是不可能的 [Cox, 1986]。这些都使多重继承看起来更具挑战性。因为我至少早在1982年已经开始思考多重继承问题了（2.13节），并在1984年发现了一种有效的实现技术，因此就无法再忍受这个挑战。我很怀疑这是惟一的一个例子，在这里时尚影响了事件发生的顺序。

1984年9月，我在坎特伯雷的IFIP WG2.4会议上演示了C++运算符的重载机制 [Stroustrup, 1984b]。在那里我遇到了来自奥斯陆大学的Stein Krogdahl，他刚刚完成了一个给Simula增加多重继承机制的建议书 [Krogdahl, 1984]。他的思想也变成了在C++里实现常规的多重基类的基础。他和我后来才知道，这个建议几乎与关于在Simula里提供多重继承的另一个思想完全一样，Ole-Johan Dahl在1966年就考虑了多重继承问题并拒绝了它，因为它将会使Simula的废料收集程序大大地复杂化 [Dahl, 1988]。

12.2 普通基类

考虑多重继承的最原始最根本的原因很简单，就是想使两个类能够以这种方式组合起来，使结果产生的类的对象的行为就像两个类中任何一个的对象 [Stroustrup, 1986]：

“使用多重继承的一个可以说是标准的例子就是：提供两个库类displayed和task，它们分别表示在一个显示管理器控制下的对象和一个调度器控制下的协作程序。而后程序员能

够以如下方式创建类

```
class my_displayed_task : public displayed, public task {
    // ...
};

class my_task : public task { // not displayed
    // ...
};

class my_displayed : public displayed { // not a task
    // ...
};
```

如果（只）使用单继承，在这三种选择方式里就只有两种是对程序员开放的。”

那时我正在担心，因为要服务于太多的需要，库类会变得过大（“充斥各种特征”）。我把多重继承看成是一种潜在的重要组织手段，利用它，围绕着几个简单的类和类间的依赖关系就能把库组织起来。上面task和displayed的例子显示出一种方法，分别用不同的类表示并发性和输出，又没有在应用程序员肩上增加过多的负担。

“歧义性在编译时就处理了：

```
class A { public: void f(); /* ... */ };
class B { public: void f(); /* ... */ };
class C : public A, public B { /* no f() ... */ };

void g()
{
    C* p;
    p->f(); // error: ambiguous
}
```

在这个方面，C++在支持多重继承方面也与面向对象的Lisp方言不同 [Stroustrup, 1987]。”

这里也拒绝了依靠某种顺序依赖关系去消解出现的歧义性，比如说偏向于A::f是因为A在基类列表里出现在B之前。究其原因，也是因为来自其他地方有关顺序依赖性的负面经验，参看11.2.2节和6.3.1节。我也拒绝除了使用虚函数之外的任何动态解析形式，认为对一个要在严格的效率约束下使用的静态类型语言而言，这种东西是完全不合适的。

12.3 虚基类

[Stroustrup, 1987] 的一段：

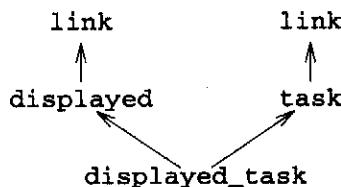
“一个类可能在一个继承的DAG（有向无环图）里出现不止一次：

```
class task : public link { /* ... */ };
class displayed : public link { /* ... */ };
class displayed_task
    : public displayed, public task { /* ... */ };
```

在这种情况下，类displayed_task的对象里就会出现两个类link的对象：task::link和displayed::link。这种情况常常也很有用处，就像在上面的情况里，表的实现要求表

里的每个元素都包含一个link元素。这就使一个displayed_task可以同时出现在displayed的表上和一个task的表上。”

从图形上也可以看出表示displayed_task所需要的子对象：



我不认为这种表的风格在任何状况中都很理想，但在它所适用的地方，这通常就是最好的方式了，因此我不想去禁止这种东西。所以C++支持上面这样的例子。按照默认方式，一个基类出现两次将用两个子对象表示。当然也存在另外一种解决办法 [Stroustrup, 1987]：

“我把这称为独立的多重继承。但无论如何，有关多重继承所提出的许多使用都假定了基类之间的依赖关系（例如，对一个窗口提供有关风格的特征选择）。这种依赖性可以用不同派生类之间共享对象的方式描述。换句话说，必须有一种方式去描述一个基类只能在最终派生的类里给出一个对象，即使它曾经作为基类出现过多次。为了将这种应用与独立的多重继承区分开，这样的基类就应该被描述为虚的：

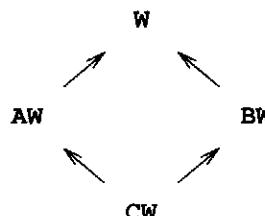
```

class AW : public virtual W { /* ... */ };
class BW : public virtual W { /* ... */ };
class CW : public AW, public BW { /* ... */ };
  
```

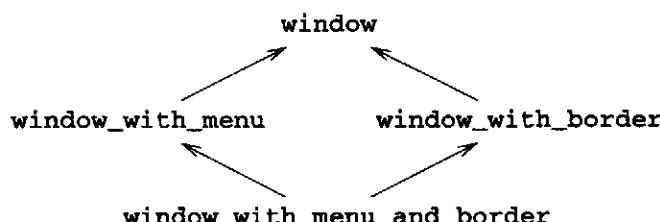
在类AW和BW之间共享了类W的惟一的一个对象；也就是说，作为由AW和BW派生CW的结果，在CW里只有一个W对象。除了只在派生类里提供一个对象之外，virtual基类在其他方面与非虚基类完全一样。

类W的“虚”是AW和BW描述的那些派生的一种性质，而不是W自身的一种性质。在一个继承DAG里，每个virtual基类总表示同一个对象。”

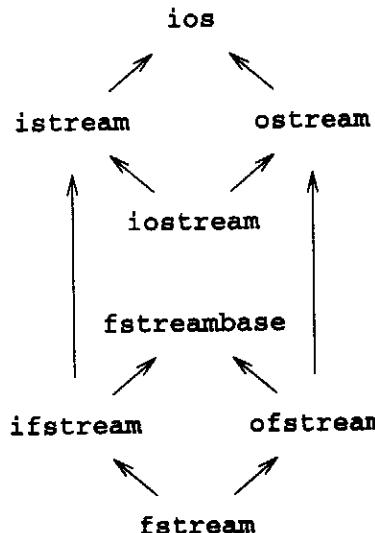
从图形看：



在实际程序里，W、AW、BW和CW能够是什么？我原来的例子是一个单一窗口系统，基于来自Lisp文献的思想：



从我的经验看，这好像有点弄巧成拙的样子。但它是基于实际的例子，也很直观。在展示中这一点非常重要。在标准iostream库里还可以找到另一些例子 [Shopiro, 1989]：



我看不出有什么理由说虚基类比普通基类更有用或是更基本，或者是反过来，所以我觉得两种东西都应该支持。选择普通基类作为默认情况，只是因为它们的实现比虚基类更节约运行的时间和空间。也因为“使用虚基类的程序设计比使用非虚基类更多一点技巧性。问题是要防止对虚类里的函数的多次调用，如果实际中不希望这样的话”[2nd]；也见12.5节。

因为实现的困难，我很想不把虚基类的概念包括到语言里。当然，我也考虑了另一种论点，那就是必须存在一种方式，以便能表示互不相关的兄弟类之间的依赖性。兄弟类只能通过一个共同的根类相互通讯，或者通过全局性数据，或者是通过显式指针。如果没有虚基类，对于公共根类的需求就会引起过度使用“统一的”基类的情况。在12.3.1节中描述的混合子风格就是这种“兄弟通讯”的一个例子。

如果要支持多重继承，那就需要包含一些这样的功能。在另一方面，我认为多重继承的那些简单而又不出奇的应用，将一个类定义为两个原本无关的类的属性之和，是最有用的东西。

虚基和虚函数

抽象类和虚基类的组合就是为了支持一种程序设计风格，它大致对应于某些Lisp系统里所使用的混合子风格。这就是说，用一个抽象类来定义界面，用若干个派生类提供实现。每个派生类（每个混合子）为完整的类（混合体）提供某些东西。按照可靠的报道，术语混合子源自MIT附近某一家冰淇淋店，他们将坚果、葡萄干、胶糖块、小甜饼等等加入冰淇淋中。

要想能使用这种风格，就需要有两条规则：

- 1) 必须能在不同的派生类里覆盖基类里的函数；否则一个实现的基本部分就只能来自单个的继承链，就像13.2.2节里的例子slist_set。
- 2) 必须能确定哪个是覆盖了虚函数的函数，并能从继承格上捕捉到所有的不一致性问题；否则我们就只能依赖于顺序相关性或者运行时的检查了。

考虑上面的例子。比如说W有虚函数f()和g()：

```
class W {
    // ...
    virtual void f();
    virtual void g();
};
```

而AW和BW各覆盖了其中的一个：

```
class AW : public virtual W {
    // ...
    void g();
};

class BW : public virtual W {
    // ...
    void f();
};

class CW : public AW, public BW, public virtual W {
    // ...
};
```

然后CW可能以如下方式被使用：

```
CW* pcw = new CW;
AW* paw = pcw;
BW* pbw = pcw;

void fff()
{
    pcw->f(); // invokes BW::f()
    pcw->g(); // invokes AW::g()

    paw->f(); // invokes BW::f() !
    pbw->g(); // invokes AW::g() !
}
```

就像对虚函数一样，被调用的将是同一个函数，与对该对象所使用的指针无关。这件事的重要性在于它将允许把不同的类加到一个公共的基类上，以便能从每个类的贡献中获益。很自然，在设计那些派生类时，应该把这一点放在心里，去组合它们的时候也需要小心，需要有关其兄弟类的知识。

允许在分支中覆盖就要求有一条规则，说明什么东西是能够接受的，哪些覆盖的组合将作为错误而拒绝之。由一个虚函数调用的必须是同一个函数，无论类的对象是如何描述的。Andrew Koenig和我发现，我们认为能够保证这些的惟一规则就是[ARM]：

“名字B::f支配A::f，如果其类B以A作为基类。如果一个名字支配另一个，在两者之间不存在歧义性，那么当存在选择的时候就使用那个支配着另一个的名字。例如：

```
class V { public: int f(); int x; };
class B : public virtual V { public: int f(); int x; };
class C : public virtual V { };
```

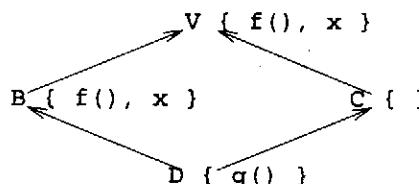
```

class D : public B, public C { void g(); };

void D::g()
{
    x++;      // ok: B::x dominates V::x
    f();      // ok: B::f() dominates V::f()
}

```

图形是：



请注意，支配关系适用于所有的名字，而不止是函数。

对于虚函数，支配规则是必需的——因为它也就是关于在一个虚调用中究竟应该执行哪个函数的规则。经验还说明，这个规则同样能很好地用于非虚函数。有一个编译器对非虚函数没有采纳支配规则，在早期使用中就引起许多程序员错误和扭曲的程序。”

从实现者的观点看，支配规则也就是一个常见的查找规则，用以确定是不是存在惟一的一个函数可以放进虚函数表。更宽松的规则将无法保证这一点，而更严格的规则将不能允许某些合理的调用。

有关抽象类的规则和支配规则能够保证，只有那些提供了完整的、一致的服务的类才能够创建对象。如果没有这些规则，程序员在使用不太简单的框架时就很难避免严重的运行错误。

12.4 对象布局模型

多重继承从两个方面使对象模型更复杂了：

- 1) 一个对象可以有多个虚函数表。
- 2) 虚函数表必须提供一种方式，以便能够去查寻提供这个虚函数的那个类所对应的那个子对象。

考虑：

```

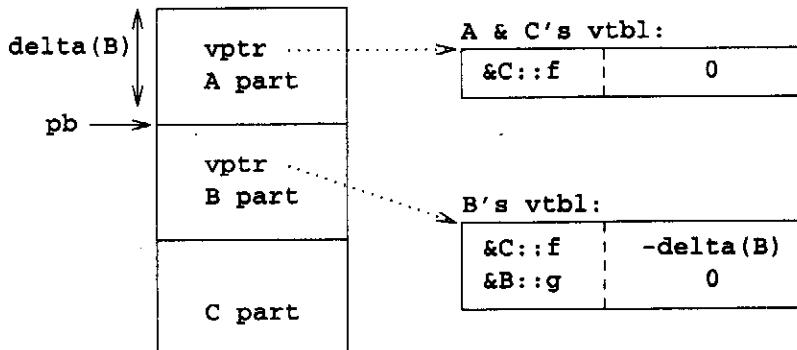
class A {
public:
    virtual void f(int);
};

class B {
public:
    virtual void f(int);
    virtual void g();
};

```

```
class C : public A, public B {
public:
    void f(int);
};
```

类C的一个对象看起来可能具有下面的样子：



在这里必须有两个vtbl，因为A和B除了可能作为一个C的组成部分外，你还可以有类A的对象和类B的对象。在得到了一个到B的指针时，你必须能去调用对应的虚函数，而不必知道到底这是一个“简单的B”，还是一个C的“B部分”，或者是在别的什么对象里面包含的B。这样，每个B都需要有一个vtbl^Θ，在所有情况下都应该以同样的方式访问。

这里的偏移量(delta)是必需的，因为一旦找到了vtbl，被调用的函数就必须针对它所定义的那个子对象进行调用。例如，对一个C对象调用g()，就要求一个指向C的B子对象的this指针。而在对一个C对象调用f()时则要求一个指向完整C对象的this指针。

有了上面所建议的布局方式，调用：

```
void ff(B* pb)
{
    pb->f(2);
}
```

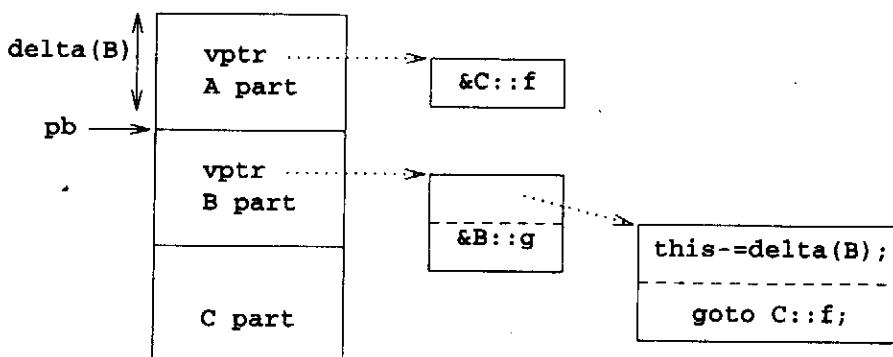
可以用类似下面的代码实现：

```
/* generated code */
vtbl_entry* vt = &pb->vtbl[index(f)];
(*vt->fct)((B*)((char*)pb+vt->delta), 2);
```

这也就是我第一次在Cfront里实现多重继承时所采纳的实现策略。它的优点就是很容易在C里表示，所以也是可移植的。生成的代码也有许多优点，在其中不包含分支，所以在高度流水线的机器体系结构中执行速度非常快。

另一种实现方式可以避免在虚函数表里为this指针保存偏移量值，而是存放一个指到要执行代码的指针。如果this不需要调整，vtbl里的指针就指向虚函数执行所对应的那个实例；当这个指针需要调整时，就让vtbl里的指针指向调整指针的代码，而后才是对应的虚函数实例。如上定义的类C这时应该有如下形式：

Θ 原书如此。为vptr之误。——译者注



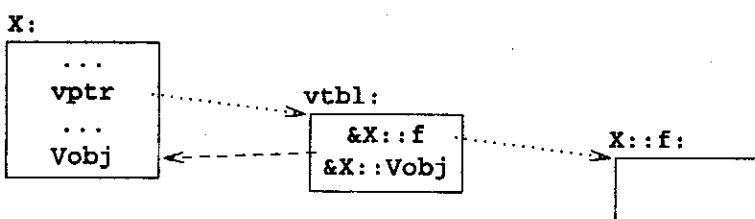
按照这种模式产生的虚函数表更紧凑一些。在偏移量是0时它也能给出更快的函数调用。请注意，在所有单继承的情况下偏移量都是0。在高度流水线的机器上，采用偏移量修改之后再改变控制的代价会很大，而这类代价具有很强的体系结构依赖性，无法给出一般的指导性意见。这种模型的缺点在于它的可移植性差一点。例如，并不是所有机器结构都允许跳进另一个函数的函数体内。

调整this指针的代码通常被称为一个小块(thunk)，这个名字至少可以追溯到Algol 60的实现，在那里采用这样的一些小代码片段实现按名字调用。

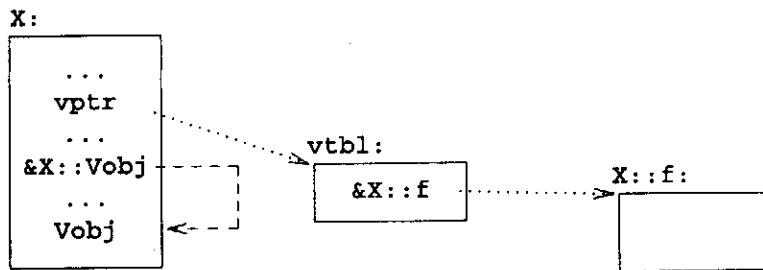
我在设计多重继承和第一次实现它的时候就已经知道了这两种实现策略。从语言设计的观点看，它们几乎就是等价的，但采用代码小块的实现方式有另一种引人注目的性质：对于只使用单继承的C++程序，它不会引起任何时间或空间代价——这恰好满足了零开销的“设计原则”(4.5节)。在我考查的一些情况中，两种实现策略所提供的性能都是可以接受的。

12.4.1 虚基布局

为什么“虚基类”被称为是virtual？通常我只给出简短的解释“这个嘛，virtual意味着一种魔力”，就继续去忙某些更紧急的事务了。但确实存在一个更好的解释，这个解释是在与Doug McIlroy的讨论中出现的，这是在C++的多重继承的第一次公众演示之前很久。一个虚函数是一个函数，你通过一个对象经过间接找到它。与此类似，在由一个虚基类派生的类中，表示这个虚类的对象并不位于某个固定位置，这样就必须通过一个间接才能够访问。还有，一个基类被定义为一个无名成员。因此，如果允许有虚数据成员，那么虚基类就该是虚数据成员的一个例子。我希望自己实现虚基类的方式确实遵循了这个解释。例如，在给定了一个具有虚基类v的类x，以及一个虚函数f之后，我们应该有：



而不是我在Cfront里所使用的那种“优化”实现：



在这两个图里，写成`&X::Vobj`的是在`X`里代表虚基类`V`的对象的偏移量。前一个模型更清晰也更具一般性。与“优化”模型相比，它多用了一点时间，也省了一点空间。

虚数据成员是人们一直建议C++做的扩充之一。典型地，某个建议者只要求有“静态虚数据”、“常量虚数据”或者甚至是“常量静态虚数据”，而不是更一般的概念。当然，根据我的经验，在人们心目中这个建议的应用就是一个：运行时的对象类型识别。存在着系统的方法得到这个东西，参见14.2节。

12.4.2 虚基和强制

虚基类是一种派生时的性质，而不是基类自身的性质，看到这种情况时人们或许会表示出一点惊讶。不管怎样，在上面讲述的对象布局模型里并没有提供足够的信息，如果只给了到对象里面的一个基类的指针，那么将无法找到派生类，因为不存在到包在外面的对象的“回向”指针。例如：

```
class A : public virtual complex { /* ... */ };
class B : public virtual complex { /* ... */ };
class C : public A, public B { /* ... */ };

void f(complex* p1, complex* p2, complex* p3)
{
    (A*)p1; // error: can't cast from virtual base
    (A*)p2; // error: can't cast from virtual base
    (A*)p3; // error: can't cast from virtual base
}
```

对于调用：

```
void g()
{
    f(new A, new B, new C);
}
```

在由`p1`所指的`complex`相对于`A`的位置未必与由`p2`所指的`complex`的相对位置相同。因此，由虚基类到派生类的强制实际要求做一个运行时的动作，这个动作需要基于存储在基类对象里的信息。在简单类的对象布局约束之下，根本就没有这种信息可以使用。

如果我对强制的疑虑更少一点，可能就会把缺乏从基类出发的强制看得更严重一些。但不管怎样，通过虚函数访问的类和只简单保存几个数据项的类通常是最好的虚基类。如果一个基类里只有数据成员，你不会把到它的指针作为整个类的代表传来传去。在另一方面，如果一个基类里面有虚函数，你就可以调用这些函数。在这两种情况下应该都不需要强制。还

有，如果你真的需要从基类强制到它的派生类，那么`dynamic_cast` (14.2.2节) 已经基于虚函数解决了这个问题。

常规的基类在一个给定派生类的每个对象里的位置都是固定的，编译程序知道这个位置。因此，一个到常规基类的指针可以强制为一个到派生类的指针，在这里没有任何特殊的问题或者开销。

如果原来就希望能将一个类显式地声明为一个“潜在的虚基类”的话，那么就可以把某些特殊规则应用于这个虚基类。例如可以加入某些信息，允许从一个“虚基”强制到由它派生的类。我没有把“虚基类”做成一种特殊类，理由是这将会迫使程序员为一个概念去定义两个不同的版本，一个作为常规的类，另一个作为虚基类。

另一种方式，我们也可以对每个类对象都增加为最一般的虚基类而需要付出的额外开销。但这样做将给不使用虚基类的应用带来严重负担，也将导致布局的不兼容问题。

正因为此，我允许任何类被用作虚基，并接受了禁止对虚基使用强制的限制。

12.5 方法组合

派生类的函数常常是基于同一函数在基类中的版本综合出来的。这一般称作方法组合，某些面向对象的语言直接支持这样做，但C++不是这样——除了建构函数、析构函数和复制函数之外。或许我应该早将`call`和`return`函数的概念救活，以模拟CLOS语言里的`:before`和`:after`方法。当然，人们早就已经在担心多重继承机制的复杂性问题，而我也很不情愿重新去打开这些老伤口。

换个角度，我也看到可以通过手工进行方法的组合。问题是，在不希望时如何避免对基类里一个函数的多次调用。这里是一种可能的风格：

```
class W {
    // ...
protected:
    void _f() { /* W's own stuff */ }
    // ...
public:
    void f() { _f(); }
    // ...
};
```

每个类提供了一个为派生类所用的保护函数`_f()`，做“这个类自己的事情”。这些类里，还提供一个公有函数`f()`作为“一般公开”使用的界面。在派生类里的`f()`通过调用`_f()`做它“自己的事情”，调用其基类的`_f()`做它们“自己的事情”：

```
class A : public virtual W {
    // ...
protected:
    void _f() { /* A's own stuff */ }
    // ...
public:
    void f() { _f(); W::_f(); }
    // ...
};
```

```

class B : public virtual W {
    // ...
protected:
    void _f() { /* B's own stuff */ }
    // ...
public:
    void f() { _f(); W::_f(); }
    // ...
};

```

特别是这种风格能允许从类W（间接）派生两次的一个类只调用W::f()一次：

```

class C : public A, public B, public virtual W {
    // ...
protected:
    void _f() { /* C's own stuff */ }
    // ...
public:
    void f() { _f(); A::_f(); B::_f(); W::_f(); }
    // ...
};

```

与自动产生组合函数相比，这种方法不那么方便，但从某个方面说它又更灵活一些。

12.6 有关多重继承的论战

C++的多重继承问题变成一场大论战 [Cargill, 1991] [Carroll, 1991] [Waldo, 1991] [Sakkinen, 1992] [Waldo, 1993]，这里面有许多原因。反对它的论点集中围绕着这个概念的实际的和想象中的复杂性，它的用途，多重继承对其他语言特征的影响，以及对构造工具的影响等等：

(1) 多重继承在那时被看作是C++的第一个主要扩充。一些C++守旧派认为这是一种不必要的花饰，一种节外生枝，很可能成为一个楔子，导致大门洞开新特征蜂拥而至进入C++。例如，在圣菲最早的那次C++会议上（7.1.2节）Tom Cargill开玩笑地，但却不是很不真实地建议说，任何人为C++提出了新特征的建议也应该建议删除一个具有类似复杂性的旧特征。他的话赢得了热烈的喝彩。我也赞成这种情绪，但也很难得出结论说没有多重继承C++就会更好些，或者说1985年的C++比它更大的1993年的化身更好一些。Jim Waldo后来附议Tom，提出了进一步的想法：对新特征的建议还应该被要求捐赠一个肾脏。这将迫使——Jim指出——人们在提出建议之前认认真真地思考，而即使是没什么见识的人至少也能提出两条建议。我也注意到，并不是每个人都对新特征那么敏感，就像人们在读杂志、读网络新闻、在讲演后听问题时所想的那样。

(2) 我的多重继承实现方式确实强加了一些负担，即使是那些只使用单继承用户也将受到影响。这违反了“你不需要为你没用的东西付出代价”的规则（4.5节），也引出了一种（错误的）印象，好像多重继承本质上就是低效的。我认为这个负担是可以接受的，因为负担很小（每次虚函数调用做一次数组访问再加上一次加法），还因为我也知道另一种实现多重继承的简单技术，能使在单继承层次上的虚函数调用根本就没有改变（12.4节）。我之所以选择“次优的”实现方式是因为它更易移植。

(3) Smalltalk不支持多重继承，而许多人将面向对象的程序设计与“好”和Smalltalk等同

看待。这种人经常这样推断：“如果Smalltalk没有它，那么多重继承一定或者是不好的，或者就是不必要的。”当然，这个结论并没有必然性。有可能Smalltalk也会从多重继承中获益，也可能它不，但这并不是问题之所在。无论如何，我很清楚的是，Smalltalk迷们建议作为多重继承替代品的几种技术根本无法用在C++这样的静态类型语言中。语言的战争多半都有点蠢，而集中在一个孤立的特征上就更愚蠢了。对多重继承的攻击实际上是错误地指向静态类型检查，或者作为保护Smalltalk，抵御想象中的攻击的一种托词，这些东西还是忘掉最好。

(4) 我对多重继承的演示 [Stroustrup, 1987] 是非常技术性的，关注点主要在实现问题，没有注意解释使用它的程序设计技术。这就使许多人得出了结论：多重继承的用处很少，并且非常难实现。我的疑问是，如果过去我对单继承的演示也采用同样方式，他们莫非也会因此得出同样的结论吗。

(5) 有些人认为多重继承从根本上说就是个坏东西，因为“它太难用，因此将导致拙劣的设计和错误百出的代码”。多重继承当然可能被过度使用，但每个令人感兴趣的语种特征也都会这样。对我来说，更重要的是看到了一些实际程序，在那里采用多重继承将产生出一种新结构，程序员都会认为它优于采用单继承的替代方式。还有，在这种地方，我无法找到任何明显的替代品能够简化程序的结构或者简化它们的维护。我的疑问是，某些关于多重继承很容易出错的断言实际上都是基于在某些语言上的经验，而那里并没有提供C++这种层次的编译时错误检查机制。

(6) 另一些人认为多重继承是一种太弱的机制，有时提出采用委托机制来取代它。委托是一种在运行中将操作前推到另一个对象去的机制 [Stroustrup, 1987]。我很喜欢委托的想法，也为C++实现了它的一个变形，想试一试。但结果是意见一致的和令人失望的：每个用户都遇到了严重的问题，最后都归咎到了他们基于委托的设计中的缺陷 (12.7节)。

(7) 也有人断言说，多重继承本身还是可以接受的，但将它放进C++就会给语言的潜在特征（例如废料收集）增加困难，也会使工具的构造（例如数据库）变得过于困难。只有时间才能告诉我们，工具方面增加的困难是否会超过由于有了多重继承而给应用的设计和实现带来的益处。

(8) 最后，也有些意见说（多数是在这个机制引进C++几年之后）多重继承本身是个好主意，但这个主意的C++版本则是错误的。这种意见可能会使“C++++”的设计者们感兴趣，但我没有发现这些建议对自己改进C++语言、与它相关的工具、以及程序设计技术的工作中有多少帮助。人们确实为他们所建议的各种改进提出实践性的证据，但这些建议都太缺乏细节，各种改进建议之间的差异太大，几乎都没有考虑从当前规则向它们的转变。

我认为——就像我那时所认为的——这些论断的根本缺陷在于把多重继承看得太严重了。多重继承不可能解决你的所有问题，但它本来就不必这样，因为它是很便宜的东西。有时有了多重继承会非常方便。Grady Booch [Booch, 1991] 表达了一种更强的情绪：“多重继承就像一顶降落伞，你并不经常需要它，但是在你需要的时候它就是最关键的了。”他的观点部分地是基于他在C++里重新实现Ada的Booch组件过程中得到的经验 (8.4.1节)。这是一个包含了容器类和关联操作的库，由Booch和Mike Vilot实现，是使用多重继承的最好例子之一 [Booch, 1990] [Booch, 1993b]。

我一直站在有关多重继承的争论之外：多重继承已经在C++里面了，不会被去掉或者做剧烈的修改；我个人时时发现多重继承非常有用；有些人强调多重继承对于他们的设计和实现

是最基本的东西；要说已经有了实在的数据或经验说明C++的多重继承在大规模使用中的价值，那还为时太早。而且，最后，我也不喜欢在无益的讨论中耗费自己的时间。

就我的评价，多重继承最成功的使用具有如下几种简单的模式：

1) 归并相互独立的，或者基本上互相独立的几个类层次；task和displayed是这方面的例子（12.2节）。

2) 界面的组合；I/O流是这方面的例子（12.3节）。

3) 从一个界面和一个实现综合出一个类；slist_set是这方面的例子（13.2.2节）。

有关多重继承的更多例子，可以在13.2.2节、14.2.7节和16.4节找到。

大部分失误出现在某些人想将某种外来的风格强加进C++的时候。特别的，CLOS的一个设计可能是基于歧义消解的线性化、在层次结构里为共享做名字匹配、用：before和：after方法建立组合操作，直接搬过来形成的东西很不令人喜欢，会增加大程序的复杂性。

12.7 委托

最早的多重继承设计是在1987年5月赫尔辛基的欧洲UNIX用户组织（EUUG）大会上展示的[Stroustrup, 1987]，其中包含了委托的概念[Agha, 1986]。

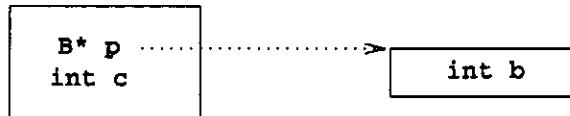
用户可以在一个类声明里描述一个指针，可以使其指向这个类的基类中的某一个。被这样指定的对象用起来恰如它就是一个代表基类的对象。例如：

```
class B { int b; void f(); };
class C : *p { B* p; int c; };
```

这里的：*p意味着使用由p指向的对象时就像它表示的是C的一个基类一样：

```
void f(C* q)
{
    q->f();      // meaning q->p->f()
}
```

类C的一个对象在C::p初始化之后，看起来就像是下面的样子：



这个概念看起来确实很有吸引力，在表达要求更多灵活性的结构方面，这样做可能可以比常规继承提供更多的东西。特别是，对委托指针进行赋值就可能在运行中实现对象的重构。其实现也非常简单，运行时间和空间效率都很理想。随后我试着为几个用户做出了一个实现。特别是Bill Hopkins对这个问题贡献了许多经验，花了许多精力。不幸的是，这个委托机制的每个用户都遭遇到严重的错误和混乱。正因为此，委托机制被从设计中和Cfront里删除了，这个Cfront就是后来发布的Release 2.0。

这里出现的两个问题是造成错误和混乱的原因：

1) 位于委托类里的函数不能覆盖被委托类里的函数。

2) 被委托函数不能使用委托类里的函数，或者以其他方式“回到”委托对象那里。

自然，这两个问题是相关的。同样也很自然，我已经考虑到这些潜在问题并提醒用户注意它们。但是这种警告并没有起作用——甚至我自己也忘记了自己的规则而落入圈套。这就使这些问题看起来不属于小污点一类的东西，无法通过教育和编译警告的组合的方式加以处理。在那时，有关的问题看起来是不可逾越的。即使我能找出一个好主意，也没有时间再用修正后的概念和实现去重复我的试验了。

回过头看，我认为其中的问题具有根本性。要解决问题1)，在一个被委托对象重新与某个委托对象建立约束时，就需要修改被委托对象的虚函数表。这一点已经超出了语言其他部分的界限，很难给出合理的定义。我们发现了这样的例子，在那里我们希望两个对象委托同一个“共享的”对象。与此类似，我们还发现了另一些例子，在那里我们需要通过一个B*委托到一个被派生类D的对象。

由于委托不是C++直接支持的东西，如果真的需要它，我们就需要能迂回地完成它。最经常遇到的情况是，一个要求采用委托的问题的解决办法涉及到灵巧指针（11.5.1节）。另一种情况是让委托类提供一个完整的界面，而后可以“手工地”将请求向前推送到某些其他对象去（11.5.2节）。

12.8 重命名

在1989年末到1990年初，好几个人都发现了由于多重继承层次结构中的名字冲突而引起的问题 [ARM]：

“通过两个基类派生一个类的方式合并两个类层次结构，如果同样的名字在两个层次结构里都使用了，而它们在不同的结构中所引用的操作又不同，那么就可能引起实际的问题。例如：

```
class Lottery {
    // ...
    virtual int draw();
};

class GraphicalObject {
    // ...
    virtual void draw();
};

class LotterySimulation
    : public Lottery, public GraphicalObject {
    // ...
};
```

在派生类LotterySimulation里我们或许想要覆盖函数Lottery::draw()和GraphicalObject::draw()，但是却想用两个不同的函数，因为draw()在两个基类里具有完全不同的意思。我们或许还想在LotterySimulation能用另外的无歧义的名字来表示由基类继承的Lottery::draw()和GraphicalObject::draw()。这个特征差一点就成为C++接受的非指定的扩充了。

这个概念的语义非常简单，实现起来也很容易，问题在于要为它找到一种适用的语法形式。下面是一种建议：

```

class LotterySimulation
    : public Lottery, public GraphicalObject {
    // ...
    virtual int l_draw() = Lottery::draw;
    virtual void go_draw() = GraphicalObject::draw;
};

```

这是以一种自然的方式扩充了纯虚函数的语法。”

经过在扩充工作组邮件反应系统里的一些讨论，以及Martin O'Riordan和我的几篇情况报告，这个建议被提交到1990年7月西雅图的标准会议上。看起来当时存在一个很大的多数，赞同将这个提议作为C++的第一个非委托的扩充。就在这时，来自Apple的Beth Crockett将委员会这辆车刹死在轨道上，提问说什么是“两周规则”。任何成员都可以将对一个建议的表决推迟到下次会议，如果这个建议在本次会议前两周还没有递交到各成员手里的话。这个规则是为了保护人们，不过于急促地去处理某些他们还没有完全理解的事情，保证他们总能有时间向同事咨询。

正如你可能想象的，Beth由于那个投票并没有立刻赢得声望。但是，她的谨慎确实很有道理，她使我们避免了一个极糟的错误。谢谢！当我们在会后重新检查这个问题时，Doug McIlroy发现，与我们的预期正相反，这个问题确实在C++内部有一种解决办法 [ARM]：

“重命名的问题可以通过对每个类引进一个额外的类的方式解决，为每个需要覆盖的虚函数使用一个具有不同名字的函数，再加上一个前向函数。例如：

```

class LLottery : public Lottery {
    virtual int l_draw() = 0;
    int draw() { return l_draw(); } // overrides
};

class GGraphicalObject : public GraphicalObject {
    virtual void go_draw() = 0;
    void draw() { go_draw(); } // overrides
};

class LotterySimulation
    : public LLottery, public GGraphicalObject {
    // ...
    int l_draw(); // overrides
    void go_draw(); // overrides
};

```

因此，专用于表示重命名的语言特征并不是必需的，除非这种解决名字冲突的需求很广泛，否则就不值得做这种语言扩充。”

在下一次会议上我演示了这种技术。在随后的讨论中，大家都认为这种名字冲突情况不会很常见，不值得增加一个语言特征。我也看到，对于新手来说合并大的类层次结构不可能成为他们日常工作。对于那些最可能做这种合并的专家而言，采用这种迂回做法与用专门的优雅语言特征并没有太大差别。

对于重命名的另一个附加的也是更具普遍性的反对意见，就是我特别不喜欢在维护代码时追寻别名的链。如果我看到拼写为f的名字实际上是在头文件里定义的g，而它又实际上是

文档里描述的h，还是你代码里的称作k的东西，那么我们确实遇到了问题。当然，这是最极端的情况，但也不出格，宏迷们已经给出了许多这样的例子。每次重命名实际上就是提出了一个用户和工具都必须理解的映射。

同义词可能是有用的，但极少会是根本性的。无论如何，为了维护清晰性和代码在不同环境中使用的统一性，应当尽可能减少同义词的使用。多一个直接支持重命名的特征就将鼓励同义词的（错误）使用。这个论据后来又重新出现，作为不提供与名字空间相关的一般重命名特征的理由。

12.9 基和成员初始式

在引进多重继承时，初始化基类及其成员的语法也必须予以扩充。例如：

```
class X : public A, public B {
    int xx;
    X(int a, int b)
        : A(a), // initialize base A
          B(b), // initialize base B
          xx(1) // initialize member xx
    {
    };
}
```

这种初始化在语法与初始化类对象的语法完全是平行的：

```
A x(1);
B y(2);
```

在此同时，初始化的顺序已经由声明的顺序确定。在C++的原来定义中，将初始化顺序放下不定义，那实际上是在损害了用户的情况下为语言实现者留下的不必要的自由度。

在大部分情况下，成员初始化的顺序并不重要，而在它起作用的大部分情况下，对顺序的依赖性实际上是表明了设计的拙劣。只有很少的情况下（不管怎样还是有），程序员绝对需要对初始化的顺序进行控制。例如，考虑在机器之间传送对象。由接收方所进行的对象重构必须正好按照传送方分解对象的相反顺序进行。除非语言明确规定构造的顺序，否则就无法保证不同提供商的编译程序编译出来的程序之间能够进行对象通讯。我记得Keith Gorlen（由于NIH库而出名，7.1.2节）向我指明了这个问题。

C++原来的定义不要求也不允许在基类初始式中指明基类。例如，给了一个类vector：

```
class vector {
    // ...
    vector(int);
};
```

我们可能派生出另一个类vec：

```
class vec : public vector {
    // ...
    vec(int, int);
};
```

vec的建构函数必须调用vector的建构函数。例如：

```
vec::vec(int low, int high)
```

```
: (high-low-1) // argument for base class constructor
{
    // ...
}
```

多年来，这种记法引起了许多混乱。

在2.0里要求明显给出基类的名字，这就使它变得相当清楚了，即使是新手也容易看出事情究竟是什么样子：

```
vec::vec(int low, int high) : vector(high-low-1)
{
    // ...
}
```

我现在认为，原来的语法作为记法的一种典型情况是合逻辑的、最小的，但是也过于紧凑了。采用了新的语法之后，在教授初始化时遇到的问题就完全消失了。

老风格的基类初始式还将保留一段时间，作为一个转变期。它只能在单继承的情况下使用，因为在其他情况中是有歧义的。

第13章 类概念的精炼

请说出你的意思，
简单而直接。

——Brian Kernighan

抽象类——虚函数和建构函数——const成员函数——const概念的精炼——静态成员函数——嵌套的类——关于inherited::建议——放松覆盖规则——多重方法——保护成员——虚函数表分配——到成员的指针

13.1 引言

由于类在C++里的中心地位，我源源不断地接到一些请求，要求对类概念进行修改和扩充。几乎所有的修改都必须拒绝，以保护现存的代码。大部分有关扩充的建议也都被拒绝了，因为不必要、不实际、与语言的其他部分不匹配，或者简单地说就是“现在处理起来太困难”。在这里我将给出若干有关精炼的例子，这些都是我觉得非常基本的，需要考虑其中的细节，在大多数情况下是可以接受的。这里的核心问题是如何使类的概念变得足够灵活，以便使各种技术都能够在类型系统的范围内表达，而不去使用强制或者其他低级结构。

13.2 抽象类

在Release 2.0发布前，最后加上去的概念就是抽象类。对一个发布的最后修改通常绝不会流行，对即将发售的东西的定义做最后修改的情况可能更差。我的印象是，在我强调这个特征时，当时的几个管理成员都认为我丢掉了与真实世界的联系。幸运的是Barbara Moo愿意支持我关于抽象类是如何重要的主张，说它们应该现在就发布出去，而不是再推迟一年或者更多的时间。

一个抽象类表示了一个界面。直接支持抽象类将能够：

- 有助于捕捉由于混淆了类作为界面的角色和它们表示对象的角色而引起的错误。
- 支持一种基于将界面的描述和实现区分开来的设计风格。

13.2.1 为处理错误而用的抽象类

抽象类直接处理了一种错误的根源 [Stroustrup, 1989b]：

“静态类型检查的一个目的就是在程序运行之前检查错误和不一致性。人们注意到，很大的一类可检查错误逃过了C++的检查。使事情雪上加霜的是，语言实际上迫使程序员去写额外的代码，生成更大的程序，这些进一步促使上述的情况发生。

考虑经典的有关“形状”的例子。在这里我们必须首先声明一个类shape，以便表示形状的最一般概念，在这个类里需要两个虚函数rotate()和draw()。很自然，根本就不会有类shape的对象，只会有特殊形状的对象。不幸的是C++并没有提供一种方式来直接表达这

个简单的概念。

C++规则规定虚函数，例如rotate()和draw()等，都必须在它们最先声明的类里定义。提出这个要求，是为了保证传统的连接程序可以用于C++程序的连接，并保证不会出现调用没有定义的虚函数的情况。所以程序员需要写像下面这样的东西：

```
class shape {
    point center;
    color col;
    // ...
public:
    point where() { return center; }
    void move(point p) { center=p; draw(); }
    virtual void rotate(int)
        { error("cannot rotate"); abort(); }
    virtual void draw()
        { error("cannot draw"); abort(); }
    // ...
};
```

这就保证了对一些情况将产生运行错误，例如在一个由shape派生的类里忘记定义draw()函数的简单错误，或者想建立一个“普遍的”shape并企图去使用它的愚蠢错误。即使是没有这些错误，存储器中也可能到处散布着像shape这样的类产生的不必要的虚函数表，或者像rotate()和draw()这样永远都不会调用的函数。这种东西造成的开销也可能变得很可观。

解决的方案就是允许用户简单地说某个虚函数并没有一个定义，也就是说，它是一个“纯的虚函数”。要做到这点只需要写初始式=0：

```
class shape {
    point center;
    color col;
    // ...
public:
    point where() { return center; }
    void move(point p) { center=p; draw(); }
    virtual void rotate(int) = 0; // pure virtual
    virtual void draw() = 0; // pure virtual
    // ...
};
```

带有一个或几个纯虚函数的类是一个抽象类。抽象类只能用作其他类的基类，特别是不可能建立抽象类的对象。从一个抽象类派生的类必须或者是给其基类的纯虚函数提供定义，或者是重新将它们定义为纯虚函数。

选择纯虚函数的概念，是想把将一个类声明为抽象类的思想明确化，选择性地定义函数是一种更灵活得多的方式。”

正如上面所显示的，在原来的C++里也可能表示抽象类的概念，只是要涉及到多做一些人们不愿意做的工作。也有一些人将它理解为一个重要的问题（如，参见[Johnson, 1989]）。当然，并不是直到2.0发布日期的前几周我才翻然醒悟，知道在C++团体中只有很少数人真正理解了这个概念。进一步说，我还认识到，对抽象类概念缺乏理解也是许多人在自己的设计中

遇到的许多问题的根源。

13.2.2 抽象类型

关于C++有一种不断听到的意见：私用数据也必须被包含在类的声明中，这实际上导致一旦某个类的私用数据改变了，使用这个类的代码也必须重新编译。这种抱怨的常见表达形式是“在C++里抽象的类型并不真正就是抽象的”以及“数据并没有被真正地隐藏起来”。我一直没有认识到，许多人这样认为，是因为当他们遇到可能把一个对象的表示放在某个类声明的私用部分时，他们实际上就必然把它放到了那里。这里的错误太明显了（这也正是我多年来一直没有指出这个问题的原因）。如果你不希望在一个类里出现某一个表示，那么就不要将它放到那里！请换个方式，把有关表示的描述推迟到某个派生类中。抽象类的概念使这件事变得非常清楚。例如可以像下面这样定义T指针的集合（set）：

```
class set {
public:
    virtual void insert(T*) =0;
    virtual void remove(T*) =0;

    virtual int is_member(T*) =0;

    virtual T* first() =0;
    virtual T* next() =0;

    virtual ~set() { }
};
```

这个定义给人们提供了有关如何使用set的所有信息。更重要的是，在这个上下文中并没有包含任何有关表示或者实现的细节。只有实际建立set对象的人才需要知道set是如何表示的。例如，给出了：

```
class slist_set : public set, private slist {
    slink* current_elem;
public:
    void insert(T* );
    void remove(T* );

    int is_member(T* );

    T* first();
    T* next();

    slist_set() : slist(), current_elem(0) { }
};
```

而后我们就可以建立slist_set的对象，它可以由那些根本不知道slist_set类的set用户们使用。例如：

```
void user1(set& s)
{
    for (T* p = s.first(); p; p=s.next()) {
        // use p
```

```

    }

void user2()
{
    slist_set ss;
    // ...
    user1(ss);
}

```

最重要的是，抽象set类的一个用户，例如user1()，可以在不必包含定义slist_set的头文件以及如slist_set这样的类定义的方式下编译，而这些实际上是它所依赖的。

如上所述，编译时将能捕捉到所有建立抽象类对象的企图。例如：

```

void f(set& s1)    // fine
{
    set s2;        // error: declaration of object
                    //           of abstract class set.
    set* p = 0;    // fine
    set& s3 = s1; // fine
}

```

抽象类概念的重要性在于它能使人对于用户和实现者做更清晰的划分，能做得比没有它的时候更好。一个抽象类就是一个纯粹的界面，对应的实现是通过由它派生的类提供的。这可以用于限制在做修改后需要重新编译的范围，也可以限制编译一部分代码所需要的信息量。通过降低用户和实现者之间的偶合度，抽象类为抱怨太长编译时间的人们提供了一种解决办法。它也能服务于库的提供商，他们自然特别关心库实现的修改对用户的影响。我看到过很大的系统，由于将抽象类的概念引进到子系统的界面里，编译时间缩短到原来的大约十分之一。我也曾在 [Stroustrup, 1986b] 里不大成功地试着解释这个概念。有了支持抽象类概念的明确的语言特征之后，我在 [2nd] 里做的就成功得多了。

13.2.3 语法

古怪的=0语法形式是从许多明显的选择（例如引进关键字pure或abstract）中挑出来的，因为那时我觉得不可能再让人接受一个新关键字。如果我建议关键字pure，Release 2.0的发布就不会包含抽象类。在这种更好的语法和抽象类的选择中我挑选了抽象类。与其说接受危险的拖延并挑起一场有关pure的战争，我还是采用了C和C++的传统习惯，用0表示“不在那里”。这个=0的语法形式也符合我的一个观点：函数体可以看成对一个函数的初始式；也符合另一个（过于简单化，倒也常常合适的）观点，认为虚函数的集合是实现为一个函数指针的向量（3.5.1节）。事实上，将=0实现为在vtbl里放一个0并不是最好的方式。我的实现是在vtbl里放一个到函数

pure_virtual_called

的指针，可以将这个函数定义为提供某种合理的运行中的错误信息。

我选择的是将个别的函数描述为纯虚的方式，没有采用将完整的类声明为抽象的形式，这是因为纯虚函数的概念更灵活一些。我很看重能够分阶段定义类的能力；也就是说，我发现预先定义一些纯虚函数，并把另外一些留给进一步的派生类去定义也是很有用的。

13.2.4 虚函数和建构函数

如何由基类和成员对象构造出对象(2.11.1节),这对于虚函数的工作方式有重要的影响。有时人们对这种影响中的某些东西比较困惑,甚至感到烦恼。下面让我来试着解释一下,为什么我认为C++采用目前的工作方式几乎是必然的。

1. 调用一个纯虚函数

怎样才能使最终被调用的是一个虚函数——而不是在派生类中覆盖它的函数?一个抽象类的对象只能作为其他对象的基而存在。一旦派生类的对象被建构起来,纯虚函数就已经被来自派生类的覆盖函数定义好了。当然,在建构过程中,也可能出现抽象类自己的建构函数错误地调用纯虚函数的问题:

```
class A {
public:
    virtual void f() = 0;
    void g();
    A();
};

A::A()
{
    f();      // error: pure virtual function called
    g();      // looks innocent enough
}
```

对A::f()的这类非法调用很容易由编译程序检查出来。当然,A::g()也可能被定义为如下形式:

```
void A::g() { f(); }
```

这还可能出现在另外的某个编译单位里。在这种情况下,只有那些真正做跨编译单位分析的编译程序才能检查出错误。另一种方式就是产生一个运行错误。

2. 基类在先的建构

与允许产生运行时错误的设计相比,我特别喜欢那些根本不开放这种可能性的设计。不管怎么样,我无法看到能使程序设计具有绝对安全性的可能性。特别的,建构函数是要建立起一个环境,使其他成员函数在其中操作(2.11.1节)。在这个环境的构筑过程中,程序员必须意识到这时能够保证的东西是非常少的。考虑下面可能引起混乱的例子:

```
class B {
public:
    int b;
    virtual void f();
    void g();
    // ...
    B();
};

class D : public B {
public:
    X x;
    void f();
```

```

// ...
D();
};

B::B()
{
    b++;    // undefined: B::b isn't yet initialized.
    f();    // calls: B::f(); not D::f().
}

```

编译程序很容易对这两个潜在问题提出警告。如果你真的希望调用B自己的f()，那就应该将它明确地写成B::f()。

这个建构函数的行为方式与写常规成员函数可能方式成鲜明的对比，因为常规成员函数可以依靠建构函数的正确行为：

```

void B::g()
{
    b++;    // fine, since B::b is a member
            // B::B should have initialized it.
    f();    // calls: D::f() if B::g is called for a D.
}

```

当一个调用出自某个D的B部分时，在B::B()和B::g()里的f()调用的是不同函数，这可能使新手感到很吃惊。

3. 换个方式会怎样？

无论如何，让我们现在来考虑另一种选择所隐含的东西，现在让对虚函数的每个调用都使用覆盖函数：

```

void D::f()
{
    // operation relying on D::X having been properly
    // initialized by D::D
}

```

如果在建构过程中也可以调用覆盖函数，那么虚函数将无法依靠建构函数的正确初始化。因此，在写一个覆盖函数时，就必须在某种程度上按照通常保留给建构函数的（偏执的）方式去做。实际上这将使写覆盖函数时遇到的情况比写建构函数时还要糟，因为相对而言你在建构函数里比较容易确定什么已经做了初始化，什么还没有做。如果不能保证一定运行过建构函数，写覆盖函数的人就只有两种选择了：

- 1) 简单地希望/假定所有必须的初始化都已经做过。
- 2) 设法进行自我保护，抵御未经初始化的基或者成员。

第一种选择将使建构函数不再有吸引力了。第二种选择则是真正无法处理的东西，因为你没有运行时能用到任意的变量上的检查手段，无法判断它是否已经做过初始化。

```

void D::f() // nightmare (not C++)
{
    if (base_initialized) {
        // operation relying on D::X having
        // been initialized by D::D
    }
}

```

```

    }
    else {
        // do what can be done without relying
        // on D::X having been initialized
    }
}

```

所以，如果让建构函数去调用覆盖函数，建构函数的用途将受到严重的限制，以至于使我们无法合理地编写覆盖函数。

在这里。基本的设计要点是，直到对一个对象的建构函数的运行结束之前，这个对象就一直像一个正在建造之中的建筑物：你必须忍受结构没有完工所带来的各种不便，常常需要依靠临时性的脚手架，必须时时当心在与危险环境相处时的各种问题。一旦建构函数返回，编译程序和用户就都可以假定构造完成的对象能够使用了。

13.3 const成员函数

在Cfront 1.0里，“常”的概念并没有完全施行，在收紧实现时我们就在语言定义里发现了一些漏洞。我们需要一种方法，使程序员可以说明哪些成员函数将更新其对象的状态，而哪些则不更新：

```

class X {
    int aa;
public:
    void update() { aa++; }
    int value() const { return aa; }
    void cheat() const { aa++; } // error: *this is const
};

```

被声明为const的成员函数，如X::value()，被称作const成员函数，并保证不会修改对象的值。const成员函数可以用于const对象和非const对象，而非const成员函数，如X::update()，就只能用于非const对象：

```

int g(X o1, const X& o2) {
{
    o1.update();      // fine
    o2.update();      // error: o2 is const
    return o1.value() + o2.value(); // fine
}

```

从技术上说，得到这种行为的方式就是让X的非const成员函数里的this指针指向X，而让其const成员函数里的this指针指向const X。

由于const成员函数与非const成员函数间存在这种差异，这使我们可以在逻辑上区分修改对象状态的函数和不这样做的函数，C++里无法直接描述这个区分。const成员函数以及其他一些语言特征从Estes Park的实现者研讨会（7.1.2节）的讨论中获益甚多。

13.3.1 强制去掉const

如常，C++关心的是检查偶然的错误，而不是防止刻意的欺骗。对我来说，这也就意味着可以允许一个函数通过“强制去掉const”来做出一个“骗局”。我们不认为编译系统有责任去防止程序员明确地做突破类型系统的事情。例如 [Stroustrup, 1992b]：

“让一些对象在用户看起来是常量，而事实上它们的状态也可以改变，这种东西偶尔也是有用的。这样的类可以通过显式的强制写出来：

```
class XX {
    int a;
    int calls_of_f;
    int f() const { ((XX*)this)->calls_of_f++; return a; }
    // ...
};
```

明确的类型转换就是要指明有些东西不很对头。改变一个const对象的状态可能不大靠得住，在某些上下文中很容易出错。如果对象位于只读存储器里，那么就根本就无法工作。一般说，更好的方式是将这种对象的可变部分描述为另一个独立对象：

```
class XXX {
    int a;
    int& calls_of_f;
    int f() const { calls_of_f++; return a; }
    // ...
    XXX() : calls_of_f(*new int) { /* ... */ }
    ~XXX() { delete &calls_of_f; /* ... */ }
    // ...
};
```

这正反映了const的基本目的，它是作为一种特殊的界面，而不是为帮助优化程序。也同时说明了另一个观点，尽管这种自由/灵活性偶尔也有用处，但它也可能被错误使用。”

通过引进const_cast (14.3.4节)，使程序员能够区分有意的“强制去掉const”的强制与有意做其他类型操作的强制。

13.3.2 const定义的精炼

为了保证某些（并不是全部）const对象能够被放进只读存储器（ROM）里，我原来采纳了这样的一个规则：任何具有建构函数的对象（它需要做运行时的初始化）都不能放进ROM，其他的const可以放。这种做法与我对什么能够做初始化，怎样做以及什么时候做的长期关注有密切关系。C++语言提供了静态（连接时的）初始化和动态（运行时的）初始化（3.11.4节）。这个规则既允许对const对象做动态的初始化，也允许对不需要做动态初始化的对象使用ROM。后一种情况的典型例子是简单对象的大数组，例如YACC的分析表。

将const概念与建构函数联系起来也是一种折衷，考虑了我对const的理想与可用硬件的现实，以及应该相信程序员在写明显的类型转换时知道自己正在做什么的观点。在Jerry Schwarz的推动下，这个规则现在已经被另一个更接近我原来理想的规则取代了。将一个对象声明为const，就是认为它具有从其建构函数完成到析构函数开始之间的不变性。在这两点之间对这个对象进行写入，其结果应该认为是无定义。

我还记得在开始设计const机制时提出过这样的论点，当时说理想的const应该是这样的一种对象，直到其建构函数完成之前它都是可以写的，而后通过某种硬件的法术就变成了只读的，而最后到析构函数的入口点它又重新变成可以写的。你可以设想一种实际上就是这样工作的带标记的系统结构，对于这种实现，如果某人企图向定义为const的对象做写入就

会引起一个运行错误。在另一方面，人则可以去写一个本身并没有定义为const，但却是经过const指针或者引用传递过来的对象。在这两种情况下，用户都必须首先强制去掉这个对象的const。这个观点意味着强制去掉一个原来就定义为const的对象的const，而后对它做写操作最好的情况就是无定义；而对一个原来并没有定义为const的对象同样做这些事情则是合法的，有清楚定义的。

请注意，按照这个精炼了的规则，const的意义将不仅依赖于这个类型是否有建构函数；从原则上说任何类都可能有建构函数。任何被声明为const的对象都可以放进ROM，放到代码段里，或者通过存储控制进行保护，等等，以保证它在接受了初始值之后不再发生变化。这种保护并不是必须要求的东西，无论如何当前的系统还不能保护每个const，使它们避免任何形式的堕落。

对一个实现而言，在如何处理const上它还是可以有很大程度的变化。让废料收集程序或者数据库系统修改一个const对象的值（例如将它移到磁盘或者移回来）不存在任何逻辑问题，只要能保证对于用户而言这个对象并没有变化。

13.3.3 易变性与强制

有些人还是特别讨厌强制去掉const，因为它是一个强制，甚至更因为这种东西并不保证对所有情况都能工作。那么我们怎样才能既不用强制写出一个像13.3.1节里xx那样的类，又不涉及像在类xxx里那样的间接呢？Thomas Ngo建议说，应该能描述一种绝不应被认为是const的成员，即使它是某个const对象的成员时也是这样。这个建议在委员会里踢来踢去了许多年，直到Jerry Schwarz的成功拥护，使它的一个变形被接受了。初始建议提出用~const作为“绝不能是const”的记法。甚至这个概念的一些拥护者也认为这个记法太难看，所以把关键字mutable（易变性）引进建议里，被ANSI/ISO委员会接受：

```
class XXX {
    int a;
    mutable int cnt; // cnt will never be const
public:
    int f() const { cnt++; return a; }

    // ...
};

XXX var;           // var.cnt is writable (of course)

const XXX const; // const.cnt is writable because
                  // XXX::cnt is declared mutable
```

从某种意义上说，这个概念还没有试验过。它确实能减少实际系统中对强制的需要，但是并不像一些人认为的那么有效。Dag Brück和其他一些人审查了数量可观的实际代码，查看哪些强制实际上是强制去掉const，哪些可以通过使用mutable而消除。这个研究证实了一个结论，一般地说，“强制去掉const”不可能完全消除（14.3.4节），用了mutable后可以从没有它时出现的所有“强制去掉const”中消去不到一半。使用mutable的益处看起来与程序设计风格有密切关系。在有些情况中借助于mutable能够消去所有的强制，而对另一些情况则一个强制都消不掉。

有些人曾经表述过这样的希望，希望修改后的const的概念和mutable在一起可以为新的重要优化打开大门。看起来情况不像是这样。获益主要还是在代码的清晰性、能够预先计算出值因此可以放进ROM的对象数量的增加、代码的分段特性等等。

13.4 静态成员函数

类的static数据成员是这样的一种成员，它只存在一个唯一的拷贝，而不像其他成员那样在每个对象中各有一个拷贝。因此，不需要引用特定的对象就可以访问static成员。static成员可以用于减少全局名字的数量，并且能把某个static成员逻辑上属于哪个类的问题弄明确，还能实现对这些名字的访问控制。这种特性对于库的提供商是非常重要的，因为它能够防止对全局名字空间的污染，并可以简化库代码的书写，使同时使用多个库变得更加安全。

这些理由除了适用于对象外也适用于函数。事实上，库的提供商最希望进行非全局化的大部分都是函数的名字。我看到过一些不能移植的代码，例如((X*)0)->f()，实际上就是想模拟static成员函数。这种计谋是个时间炸弹，因为或迟或早某个人可能将以这种方式调用的某个f()声明为virtual。而后这个调用就会令人毛骨悚然的失败，因为在地址0根本不存在一个X对象。即使当f()不是virtual时，在某些动态连接的实现中这种调用也可能失败。

在1987年赫尔辛基的EUUG（欧洲UNIX用户组织）我所给的一个讲座里，Martin O'Riordan向我指出，static成员函数是一种很清晰很有用的组织方式。这可能是第一次提出这个思想。Martin当时在爱尔兰的Glockenspiel工作，而后继续下去成为Microsoft C++编译系统的主要结构设计师。后来Jonathan Shapiro也拥护这个思想，并保证了它在Release 2.0的大量工作中并没有迷失道路。

一个static成员函数也是一个成员，所以它的名字是在类的作用域里，一般的访问控制同样起作用。例如：

```
class task {
    // ...
    static task* chain;
public:
    static void schedule(int);
    // ...
};
```

一个static成员声明仅仅是一个声明，它所声明的对象或者函数必须在程序里的某个地方有唯一的定义。例如：

```
task* task::chain = 0;
void task::schedule(int p) { /* ... */ }
```

由于static成员函数并不关联于任何特定对象，因而不需要用特定成员函数的语法进行调用。例如：

```
void f(int priority)
{
    // ...
    task::schedule(priority);
```

```
// ...
}
```

在某些情况下，类被简单地当做一种作用域使用，把不放进来就是全局的名字放入其中，作为它的static成员，以便使这些名字不会污染全局的名字空间。这也是名字空间概念的一个起源（第17章）。

static成员函数也是从Estes Park的实现者研讨会（7.1.2节）的讨论中获益甚多的语言特征之一。

13.5 嵌套的类

正如在13.2节里所说的，嵌套类是由ARM重新引进C++的。这就使作用域规则更加规范，并改进了信息局部化的能力。我们现在能够写：

```
class String {
    class Rep {
        // ...
    };
    Rep* p; // String is a handle to Rep
    static int count;
    // ...
public:
    char& operator[](int i);
    // ...
};
```

这使Rep类成为局部的东西。不幸的是，这样做也使放在类声明里的信息量进一步增加，由此导致编译时间延长，也使重新编译更加频繁。人们把过多的兴趣放到嵌套类上，偶尔也把过多经常需要改变的信息放了进去。在许多情况下，有关类（例如String）的用户对于这些信息实际上并不感兴趣，因此这些信息应该与其他类似的信息一起安置在别的地方。Tony Hansen建议允许嵌套类的前向声明，按照与成员函数和static成员一样的方式处理：

```
// file String.h (the interface):
class String {
    class Rep;
    Rep* p; // String is a handle to Rep
    static int count;
    // ...
public:
    char& operator[](int i);
    // ...
};

// file String.c (the implementation):
class String::Rep {
    // ...
};

static int String::count = 1;
```

```
char& String::operator[](int i)
{
    // ...
}
```

这个扩充已经被接受了，因为它非常简单地纠正了一个被忽视的问题。当然，它所支持的技术也不应该低估。人们还是在不断地将各种没有必要的东西装入他们的头文件，从而受到过长编译时间的伤害。因此，每一种能够帮助减少用户和实现者之间不必要的偶合的技术都是非常重要的。

13.6 Inherited::

在很早的一次标准化会议上Dag Brück递交了一份扩充建议，不少人都表示对它很有兴趣 [Stroustrup, 1992b]：

“大量的类层次结构都是以‘递增方式’建立起来的，通过在派生的类里增加函数，扩充基类的行为。很典型的情况是在派生类里的函数调用基类的函数，而后再执行一些附加的操作：

```
struct A { virtual void handle(int); };
struct D : A { void handle(int); };
void D::handle(int i)
{
    A::handle(i);
    // other stuff
}
```

对handle()的调用必须量化（写A::...——译者），以避免出现递归的循环。按照所建议的扩充，这个例子可以写成下面的形式：

```
void D::handle(int i)
{
    inherited::handle(i);
    // other stuff
}
```

通过关键字inherited进行量化可以看成是用类名字量化的一种推广形式，它能解决一些通过类名字量化可能引起的问题，对于类库的维护是非常重要的。”

我在设计C++时早就考虑过这个问题，但已经拒绝了它，转到采用基类名字的量化方式，因为这种量化能够处理多重继承，采用inherited::当然无法解决。但无论如何，Dag研究了两种模式组合的问题，这样将能处理所有的问题又不会引进漏洞：

“大部分类层次结构都是在心里想着单继承的情况下开发出来的。如果我们改变了继承树，使类D转而由类A和B派生，我们将得到：

```
struct A { virtual void handle(int); };
struct B { virtual void handle(int); };
struct D : A, B { void handle(int); };

void D::handle(int i)
```

```

    {
        A::handle(i);           // unambiguous
        inherited::handle(i); // ambiguous
    }

```

在这个情况下A::handle()是合法的C++, 同时也可能是错的。inherited::handle()用在这里有歧义, 在编译时将产生一个错误信息。我认为这种行为正是我们想要的, 因为它迫使合并两个类层次结构的人去解决这个歧义性问题。从另一方面看, 这个例子也说明多重继承可能进一步限制inherited的使用。”

由于这些论据, 以及说明了其中有关细节的细致的纸面工作, 我终于被说服了。这个建议明显是非常有用的, 很容易理解, 也很容易实现。它也有真正的实际经验基础, 因为它的一个变形已经由Apple基于他们在Object Pascal的基础上实现了。它也是Smalltalk里super机制的一种变形。

经过在委员会里对这个建议的最后讨论, Dag志愿把它提供出来, 作为教科书上的一个例子, 作为一个好思想但是又不应该被标准接受的例子 [Stroustrup, 1992b]:

“这个建议有很好的论据——就像大部分建议一样——在委员会里有许多人是它的专家, 也有许多实际经验。对于这个案例, Apple的代表已经实现了本建议。在讨论中我们很快就达成统一意见: 这个建议不存在重大缺陷。特别是与在这个方向上以前的一些建议不同(如某些早在1986年讨论多重继承时就出现过的建议), 它正确地处理了由于使用多重继承而引起的歧义性问题。我们也同意, 这个建议在实现上非常简单, 应该能从正面对程序员有所帮助。

请注意, 这些对于被接受而言还是不够充分的。我们知道十多个与此类似的小改进, 还有十来个更大一些的东西。如果我们接受了所有这些东西, 语言将由于其重量而沉没(请记住Vasa! ^Θ)。当时我们还不知道这个建议能否通过, 因为事情还在讨论之中, Micheal Tiemann走进来, 嘟囔着, 大概是“我们不需要这个扩充, 因为我们早就能够写像这样的代码了。”当“我们当然不能!”的嗡嗡声减退下去之后, Micheal告诉我们应该如何做:

```

class foreman : public employee {
    typedef employee inherited;
    // ...
    void print();
};

class manager : public foreman {
    typedef foreman inherited;
    // ...
    void print();
};

void manager::print()
{
    inherited::print();
    // ...
}

```

^Θ 参见第6章6.4节作者对古代瑞典军舰Vasa命运的说明。——译者注

有关这个例子的进一步讨论见 [2nd, 205页]。我们没有注意到的是，将嵌套的类重新引进C++里已经打开了控制作用域的可能性，使我们能够像处理其他名字一样去解析类型的名子了。

有了这种技术之后，我们决定还是把自己的努力放到其他的标准化工作上去。将inherited::作为一种内部功能的获益并不足以超过程序员用现有机制能够得到的利益。因此我们决定不把inherited::作为我们能够接受的极少数C++扩充之一。”

13.7 放松覆盖规则

请考虑写一个函数，让它返回某对象的一个拷贝。假定存在着复制建构函数，这件事做起来就非常简单：

```
class B {
public:
    virtual B* clone() { return new B(*this); }
    // ...
};
```

现在，在任何由B派生的类中如果覆盖了B::clone，这个类的对象也都能正确地克隆了。例如：

```
class D : public B {
public:
    // old rule:
    // clone() must return a B* to override B::clone():
    B* clone() { return new D(*this); }

    void h();
    // ...
};

void f(B* pb, D* pd)
{
    B* pb1 = pb->clone();
    B* pb2 = pd->clone(); // pb2 points to a D
    // ...
}
```

不幸的是，pd指向的是一个D（或者某个由D派生的东西）的事实却丢掉了：

```
void g(D* pd)
{
    B* pb1 = pd->clone(); // ok
    D* pd1 = pd->clone(); // error: clone() returns a B*
    pd->clone()->h(); // error: clone() returns a B*

    // ugly workarounds:

    D* pd2 = (D*)pd->clone();
    ((D*)pd->clone())->h();
}
```

在实际代码中，这种情况已被证明是很讨厌的事情。有些人研究了有关的规则，在这里覆盖函数的类型必须与被覆盖函数的类型完全相同，这个规则实际上可以放松而不会在类型系统中打开漏洞，也不会给实现带来严重的复杂性。例如，下面的例子应该可以允许：

```

class D : public B {
public:
    // note, clone() returns a D*:
    D* clone() { return new D(*this); }

    void h();
    // ...
};

void gg(B* pb, D* pd)
{
    B* pbl = pd->clone(); // ok
    D* pd1 = pd->clone(); // ok
    pd->clone()->h(); // ok

    D* pd2 = pb->clone(); // error (as always)
    pb->clone()->h(); // error (as always)
}

```

这个扩充最先是由Alan Snyder建议的，碰巧也是第一个正式推荐给委员会的建议。它在1992年被接受。

我们在接受它之前提出了两个问题：

- 1) 这里是否存在严重的实现问题（比如说，在多重继承或者到成员的指针的领域）？
- 2) 在所有可能的处理覆盖函数返回类型的转换中，哪些（如果有的话）是值得做的？

就个人来说，我对问题1)并不担心，因为我想我知道如何一般地去实现这种放松的规则。但Martin O'Riordan确实担心，并写了东西要求委员会演示实现的细节。

我的主要问题是想确定这个放松是否值得做，针对哪样的一集转换去做？对派生类的对象调用一个虚函数，又需要对具有这个派生类的返回值执行某种操作，这种情况究竟有多么常见？有几个人，特别是John Bruns和Bill Gibbons，坚持认为这种需求非常广泛，并不局限于少量像clone这样的计算机科学的例子。最后使我认清这个问题的数据是Ted Goldstein的观察，在他参与的Sun的一个几百万行的系统里，所有的强制中大约有三分之二是某种迂回，通过这种放松的覆盖规则都可以删除掉。换句话说，我发现最吸引人的东西是这种放松将使人能在类型系统内做一些重要的事情，而不必使用强制。这就把放松覆盖函数返回类型的问题带入了我努力的主要方向：使C++程序设计更安全、更简单，也更具说明性。放松覆盖规则不仅能删除掉许多常规的强制，同时也能清除掉去错误地使用新的动态强制的一种诱惑，该机制是与这个放松规则同时讨论的（14.2.3节）。

经过对各种可能选择的一些讨论之后，我们决定允许用D*覆盖B*以及用D&覆盖B&，只要B是D的一个可以访问的基类。此外，如果安全的话还可以加上或者去掉const。我们还决定不放松另外一些在技术上也可以处理的转换，例如从D到可以访问的基类B，从D到一个存在着转换的X，从int*到void*，从double到int等等。我们认为，允许这些覆盖产生的转换是得不偿失的，这样既会增加实现的代价，又存在迷惑用户的潜在可能性。

放松参数规则

我过去一直对放松有关返回类型的覆盖规则心存疑虑，其中的一个重要原因是，按照我的经验，这个放松毫无疑问地将伴随着一个不可接受的“等价的”放松参数类型规则的建议。例如：

```
class Fig {
public:
    virtual int operator==(const Fig&);
    // ...
};

class ColFig: public Fig {
public:
    // Assume that ColFig::operator==()
    // overrides Fig::operator==()
    // (not allowed in C++).

    int operator==(const ColFig& x);
    // ...
private:
    Color col;
};

int ColFig::operator==(const ColFig& x)
{
    return col == x.col && Fig::operator==(x);
}
```

这看起来好像也说得通，使人可以写出有用的代码。例如：

```
void f(Fig& fig, ColFig& cf1, ColFig& cf2)
{
    if (fig==cf1) { // compare Figs
        // ...
    } else if (cf1==cf2) { // compare ColFigs
        // ...
    }
}
```

可惜的是，这个东西也将导致隐含地违反类型系统的情况：

```
void g(Fig& fig, ColFig& cf)
{
    if (cf==fig) { // compare what?
        // ...
    }
}
```

如果ColFig::operator==()能够覆盖Fig::operator==()，那么cf==fig就将用一个普通的Fig参数去调用ColFig::operator==()。这将造成一个大灾难，因为在ColFig::operator==()的操作里需要访问成员col，而在Fig里根本没有这个成员。如果ColFig::operator==()对它的参数写东西，结果就将造成存储的混乱。在我第一次设

计虚函数的规则时就考虑过这个情景，认定它是不可接受的。

如果允许这种覆盖，随之而来的将是对虚函数的每个参数都需要进行运行时检查。优化这些检查不会很容易。如果没有全局分析，我们将无法知道来自其他文件的某个对象是否恰好具有某个做过这种危险覆盖的类型。这种检查所造成的负担将使有关规则缺乏吸引力。还有，如果每个虚函数调用都变成一个潜在异常的发源地，用户必须为它们做好准备，这当然是无法接受的。

程序员可以采用的另一种方式是直接进行检测，看是否需要对一个派生类型的实际参数做其他处理。例如：

```
class Figure {
public:
    virtual int operator==(const Figure&);
    // ...
};

class ColFig: public Figure {
public:
    int operator==(const Figure& x);
    // ...
private:
    Color col;
};

int ColFig::operator==(const Figure& x)
{
    if (Figure::operator==(x)) {
        const ColFig* pc = dynamic_cast<const ColFig*>(&x);
        if (pc) return col == pc->col;
    }
    return 0;
}
```

采用这种方式，运行时做检查的强制`dynamic_cast` (14.2.2节) 恰好成了放松的覆盖规则的补充。放松的规则安全而又具有说明性地处理了返回类型的问题，而`dynamic_cast`运算符能够显式地比较安全地处理参数类型。

13.8 多重方法

我还反复地考虑过基于多个对象的一种虚函数调用机制，通常被称为多重方法。我拒绝多重方法时带着许多遗憾，因为我喜欢这个想法，但是却无法找到一种可以接受的形式并按照该形式接受它。考虑：

```
class Shape {
    // ...
};

class Rectangle : public Shape {
    // ...
};

class Circle : public Shape {
```

```
// ...
};
```

我们如何设计一个intersect()，使它能正确地对其两个参数调用？例如：

```
void f(Circle& c, Shape& s1, Rectangle& r, Shape& s2)
{
    intersect(r,c);
    intersect(c,r);
    intersect(c,s2);
    intersect(s1,r);
    intersect(r,s2);
    intersect(s1,c);
    intersect(s1,s2);
}
```

如果r和s分别表示Circle和Shape^Θ，我们可能需要用4个函数来实现intersect：

```
bool intersect(const Circle&, const Circle&);
bool intersect(const Circle&, const Rectangle&);
bool intersect(const Rectangle&, const Circle&);
bool intersect(const Rectangle&, const Rectangle&);
```

每个调用都能执行到正确的函数，与虚函数的方式类似。但是，正确的函数选择必须基于两个实际参数的运行时类型。就我的看法，这里的基本问题是需要找到：

- 1) 一种调用机制，它能够像虚函数所使用的表查找那样简单而有效。
- 2) 一集规则，使得歧义性消解成为一个纯粹的编译时的工作。

我并不认为这个问题是无法解决的，但也一直没有看到这个问题的压力变得足够大，以至使它上升到我的排着队的事务堆的最顶端，迫使我拿出足够长的时间去把一个解决方案的细节做出来。

我一直有一个担心，一个快速的解决办法看来需要大量的存储，用于某种虚函数表的等价物；而任何不“浪费”大量空间去拷贝表项的方法都可能很慢，其性能特征使人无法接受；或者是两个缺点同时存在。例如，对于Circle和Rectangle的例子，任何无须考虑在运行时查找被调用函数的实现方法看来都需要4个函数指针。增加另一个类Triangle后，看起来我们就需要9个函数指针了。从Circle类派生一个类Smiley，指针需要增加到16个，虽然看起来我们有可能节约最后的这7个指针，方法是对所有Smiley都直接用与Circle有关的项。

更糟的是，指向这些函数的指针数组应该与虚函数表等价，但它们无法早早地综合起来，必须等到全部程序都已经知道，也就是说，只能由连接程序产生。其中的原因是，所有的重载函数不会属于任何一个统一的类。不可能有这样一个类，恰恰是因为每个所需要的函数都依赖于两个或者更多的参数的类型。在当时这个问题是无法解决的，因为我不希望存在某种语言特征，它需要依赖于不寻常的连接程序的支持。经验告诉我，要得到这种支持恐怕还得等许多年。

另一个使我烦恼的问题是如何处理歧义性，虽然这个问题看起来是能解决的。一个明显的回答是，让多重方法的调用遵从与其他调用同样的有关歧义性的规则。无论如何，我觉得

^Θ 原文如此。从上下文看应该是Rectangle，而不是Shape。——译者注

这个回答是很含糊的，我曾一直在为调用多重方法寻找特殊的语法和规则。例如：

```
(r@s)->intersect(); // rather than intersect(r,s)
```

但这是一个死胡同。

Doug Lea曾经建议过一个更好的解决方案 [Lea, 1991]：允许明确地将参数声明为 `virtual` 的。例如：

```
bool intersect(virtual const Shape&, virtual const Shape&);
```

对于名字的匹配和参数类型上按照放松后的匹配规则都能匹配的函数，可以采用与返回类型覆盖相同的规则。例如：

```
bool intersect(const Circle&, const Rectangle&) // overrides
{
    // ...
}
```

最后，多重方法将用常规调用语法去调用，就像前面所显示的那样。

多重方法是C++的一个很有趣的“如果……怎么样”。我能按时将它设计和实现得足够好吗？它们的应用真的那么重要，值得那些相关的努力吗？哪些其他工作可以放到一旁不做，为设计和实现多重方法提供时间？从大约1985年以来，我一直在为没有提供多重方法而感到某种遗憾和内疚。比如说，我在OOPSLA上给过的唯一的一次正式报告是在一个专题讨论中，讲的是反对语言偏执和无意义的“宗教性的”语言战争 [Stroustrup, 1990]。在其中我提到CLOS中我所喜欢的一点东西，其中还强调了多重方法。

绕过多重方法

那么，在没有多重方法的情况下我们该如何写像 `instersect()` 这样的函数呢？

在引进运行时类型识别机制（14.2节）之前，在运行中对基于类型进行解析的唯一支持就是虚函数。因为我们希望根据两个参数做解析，我们大概就需要两次虚函数调用。对上面有关 `Circle` 和 `Rectangle` 的例子，调用中存在着三种可能的静态参数类型，因此我们可以提供三个虚函数：

```
class Shape {
    // ...
    virtual bool intersect(const Shape&) const =0;
    virtual bool intersect(const Rectangle&) const =0;
    virtual bool intersect(const Circle&) const =0;
};
```

在派生类里应该正确地覆盖这些虚函数：

```
class Rectangle : public Shape {
    // ...
    bool intersect(const Shape&) const;
    bool intersect(const Rectangle&) const;
    bool intersect(const Circle&) const;
};
```

这样，任何对 `instersect()` 的调用都能正确地解析为 `Circle` 或者 `Rectangle` 的适当函数。我们还必须保证，那些使用了非特定的 `Shape` 参数的函数将通过第二次虚函数调用解析到特

定函数去：

```
bool Rectangle::intersect(const Shape& s) const
{
    return s.intersect(*this); // *this is a Rectangle:
                               // resolve on s
}
bool Circle::intersect(const Shape& s) const
{
    return s.intersect(*this); // *this is a Circle:
                               // resolve on s
}
```

其他`instersect()`函数都可以简单地对两个已知类型的参数工作。请注意，只有第一个`Shape::instersect()`函数是必需的，另外两个`Shape::instersect()`函数不过是一种优化，如果在设计基类时已经知道有哪些派生类，那么就能这样做了。

这种技术称为双重发送，它第一次出现在 [Ingalls, 1986]。在C++的环境中做双重发送也有它的弱点：在一个类层次结构中添加新的类时，将需要修改已有的类。一个派生类，例如`Rectangle`，必须知道它所有的兄弟类，还要包含虚函数的完整集合。例如，增加一个新类`Triangle`就需要修改`Circle`和`Rectangle`，如果还希望继续有上面谈到的优化，那么就需要修改`Shape`：

```
class Rectangle : public Shape {
    // ...
    bool intersect(const Shape&);
    bool intersect(const Rectangle&);
    bool intersect(const Circle&);
    bool intersect(const Triangle&);
};
```

简单地说，在C++里的双重发送能够以合理的效率和合理的优雅形式，实现在层次结构中的漫游，它适合用在那些你能够修改类的声明，以便处理新加入的类，而且有关派生类不经常变化的地方。

替代技术涉及到将某种类型标识符存储在对象里面，而后基于它们去选择被调用的函数。使用`typeid()`做运行时类型识别（14.2.5节）是这类技术的一个简单例子。也可以采用维护一个数据结构，在其中保存指向有关函数的指针，并通过类型标识符访问这个结构。这种方法也有优点，在这里的基类不需要有关派生类存在的任何知识。例如，有了合适的定义以后

```
bool intersect(const Shape* s1, const Shape* s2)
{
    int i = find_index(s1.type_id(), s2.type_id());
    if (i < 0) error("bad_index");
    extern Fct_table* tbl;
    Fct f = tbl[i];
    return f(s1, s2);
}
```

就能对任何一对可能类型的参数调用到正确的函数。简而言之，这种方式实际上就是以手工方式实现了以前一直隐藏着的虚函数表。

多重方法的特殊情况都能比较简单地模拟实现，这也是导致多重方法一直不能出现在我

的工作表最上面的主要原因，因为这使我不想花足够的时间去做出它的细节。从最实际的观念上看，这个技术也就是人们在C语言里模拟虚函数所用的技术。如果用得不多，采用这种迂回方式也还是可以接受的。

13.9 保护成员

当C++被主要用作一种数据抽象程序设计语言，或者用到一大类使用继承机制的面向对象程序设计的问题时，简单的私用/公用数据隐藏模型对C++就已经很合适了。但是，如果使用了派生类，那么就存在着类的两种用户：派生类和“一般公众”。类的成员和友员函数代表着有关的用户在类对象上进行各种操作，私用/公用机制使程序员可以清楚地描述实现和一般公众之间的划分，但却没有提供某种方式以迎合派生类的特殊需要。

在Release 1.0推出后不久，Mark Linton顺便到我的办公室来了一下，提出了一个使人印象深刻的请求，要求提供第三个控制层次，以便能直接支持斯坦福大学正在开发的Interviews库（8.4.1节）中所使用的风格。我们一起揣摩，创造出单词protected以表示类里的一些成员，它们对于这个类和它的派生类“像公用的”，而对其他地方就“像私用的”。

Mark是Interviews的主要设计师。他的有说服力的争辩是基于实际经验和来自真实代码的实例。他论证说，保护数据对于设计一个高效的可扩充的X窗口工具包是最关键的东西，而可能替代保护数据的其他方式都因为低效、难以处理在线界面函数或者使数据公开等等，因而是无法接受的。保护数据（一般说是保护成员）看起来更少有害。还有，自称“纯粹”的语言，如Smalltalk也支持这种东西——保护的稍微弱一些的观念，而不是C++的private这样强的概念。我也写过一些代码，在其中把数据声明为public，就是为了让派生类里能够使用。也看到过一些代码，在那里friend观念被笨拙地错误使用，以便将访问权授予命名的派生类。

这些都是很好的论据，非常重要，它们使我确信应该允许有保护成员。当然，我是把“很好的论据”看作是在讨论程序设计时值得高度关注的东西。看起来对每个可能的语言和它的每种使用都存在“很好的证据”。我们还需要数据。如果没有数据和适当的评价试验，我们就会像那些希腊哲学家，他们确信宇宙中所有东西都是由某几种物质组成的，他们天才地争辩了几个世纪，然而却还是无法确定究竟是哪四种（或者是五种）基本物质。

大约五年之后，Mark在Interviews里禁止了保护数据成员，因为它们已经变成许多程序错误的根源：“当新用户摸不着道时就到处碰，希望自己能够知道得更好一点。”它们也使维护大大复杂化了：“现在看起来应该改变这种情况，你认为除了在这里之外还有什么人使用它吗？”Barbara Liskov在OOPSLA的主旨发言[Liskov, 1987]中给出了一个细节解释，讨论了基于protected概念进行访问控制的理论和实践性问题。按照我的经验，在把重要信息放在基类里，以便使派生类能够直接使用的问题上总存在着许多可选的方案。实际上，我对protected的关心正在于它将导致使用一个基类变得太容易，就像人们可能因为懒惰而使用全局数据一样。

幸运的是你不必在C++里使用保护数据，private是类里的默认情况，通常也是更好的选择。请注意，对于保护成员函数来说，这些反对意见中并没有什么重要的东西。但我还是认为，对于描述能在派生类里使用的操作来说，protected是一种极好的方式。

保护成员是Release 1.2引进的，保护基类最早是在ARM里描述的，Release 1.2提供了它。

回过头看，我认为protected是“好的论据”和时尚战胜了我的更好的判断和经验规则，使我接受新特征的一个例子。

13.10 改进代码生成

对于大部分用户而言，Release 2.0的最重要“特征”根本就不是某个特征，而是空间的优化。从一开始Cfront生成的代码质量就很不错。到了1992年，Cfront在一个SPARC上对一个用于评价C++编译程序的基准测试程序集生成了最快的运行代码。除了在Release 3.0里实现了[ARM, 12.1c节]所提出的返回值优化外，在Release 1.0以后由Cfront生成的代码在速度上没有重大改进。但无论如何，Release 1.0非常浪费空间，因为对于在一个编译单位里使用的所有类，这个编译单位都需要生成自己的一集虚函数表。这可能引起成兆字节的浪费。那时（大约1984年）我认为，在缺乏连接程序支持的情况下，这种浪费是不可避免的，并要求得到有关的支持。到了1987年，连接程序的支持还是没有实现。这样，我只好重新考虑有关问题并设法解决了它，采用的是一种简单的启发式方法，将一个类的虚函数表安放到它的第一个非纯虚的非在线函数的定义之后，例如：

```
class X {
public:
    virtual void f1() { /* ... */ }
    void f2();
    virtual void f3() = 0;
    virtual void f4(); // first non-inline non-pure virtual
    // ...
};

// in some file:

void X::f4() { /* ... */ }

// Cfront will place X's virtual function table here
```

我选择这个启发方法，是因为它不需要连接程序的合作。这个启发方法并不完美，因为对那些没有任何非在线虚函数的类还是会浪费空间，但是由虚函数表占用的空间已经不再是实际问题了。Andrew Koenig和Stan Lippman参与了有关这个优化的细节讨论。自然，其他C++可以，而且也确实选择了它们自己对这个问题的解决办法，以适应它们的环境和工程方面的折衷。

作为另一种选择，我们也考虑了在每个编译单位里简单地生成一个虚函数表定义，而后再用一个前连接程序删除所有多余的表，只留下一个。不管怎样，这种方法也很难做成可移植的。它当然是低效的。为什么要生成所有这些表，而只是为了后面花时间去丢掉大部分东西呢？人们当然想出了其他的技术，希望能支持自己的连接程序。

13.11 到成员的指针

开始时，在C++里没有什么办法去表述指向成员函数的指针这个概念。在一些情况下这也引发了一种欺骗类型系统的需要，例如在处理错误时，传统方式就是使用指向函数的指针。有了

```
struct S {
    int mf(char*);
```

人们会写出下面这样的代码：

```
typedef void (*PSmem)(S*,char*);  
  
PSmem m = (PSmem)&S::mf;  
  
void g(S* ps)  
{  
    m(ps,"Hello");  
}
```

这种东西只能通过到处随便散布显式强制的方式工作，首先是绝不应该这样做。这种做法还依赖于一个假设：成员函数总是按照Cfront的实现方式（2.5.2节），通过第一个参数得到它的对象指针（“对应的this指针”）。

早在1983年我就认为这是不可接受的，但又不觉得修正它是件急迫的事情。我那时认为这只是一个纯粹的技术问题，需要回答它以封住类型系统上的一个漏洞，但在实践中它并不很重要。在完成了Release 1.0的工作后，我终于找到了一点时间去塞住这个漏洞，Release 1.2实现了有关的解决方案。碰巧，把回调当作基本通讯机制的系统不久就降临人世，这就使对该问题的解决方案变成非常关键的事情了。

术语指向成员的指针很容易使人误解，因为到成员的指针实际上比一个偏移量（标识对象中一个成员的值）还要多一些东西。当然，由于我把它称作“偏移量”，人们可能已经做了错误的假设，认为到成员的指针不过是到对象里面的一个简单指标，也或许会假定能够使用某种形式的算术运算。这些可能已经造成比术语指向成员的指针更多的混乱。我选择这个术语，是因为在设计这个机制时采用的是很接近C指针的语法形式。

考虑C/C++里最辉煌的函数语法：

```
int f(char* p) { /* ... */ } // define function.  
int (*pf)(char*) = &f; // declare and initialize  
                      // pointer to function.  
int i = (*pf)("hello"); // call through pointer.
```

将S::和p->插入其中适当的位置，我就为成员函数构造出一种与此平行的东西：

```
class S {  
    // ...  
    int mf(char*);  
};  
  
int S::mf(char*p) { /* ... */ } // define member function.  
int (S::*pmf)(char*) = &S::mf; // declare and  
                                // initialize pointer to  
                                // member function.  
S* p;  
int i = (p->*pmf)("hello"); // call function through  
                            // pointer and object.
```

这种指向成员函数的指针在语义上和语法上都很有意义。我需要做的全部工作就是将它推广到数据成员，并找到一种实现策略。在[Lippman, 1988]的致谢一节中这样说：

“指向成员的指针概念的设计是与Bjarne Stroustrup和Jonathan Shapiro合作努力的结果，还有Doug McIlroy的许多很有意义的指教。Steve Dewhurst对于重新设计指向成员的指针，使之适用于多重继承的概念，做出了很大的贡献。”

那时我很喜欢说我们是发现了指向成员的指针概念，而不是我们设计了它。对2.0的大部分东西都有这种感觉。

在很长一段时间，我都认为指向数据成员的指针是为了推广而做出来的一种人造物件，而不是什么真正有用的东西。我又一次被证明是错的。特别是实践已经证明，指向数据成员的指针是一种表达C++的类布局方式的与实现无关的方式 [Hubel, 1992]。

第14章 强 制

明白事理的人
不会去改变世界。
——萧伯纳

主要和次要扩充——对运行时类型信息的需求——`dynamic_cast`——语法——哪些类型支持RTTI——从虚基出发的强制——RTTI的使用和误用——`typeid()`——类`type_info`——扩充的类型信息——一个简单的I/O对象系统——拒绝其他新强制——`static_cast`——`reinterpret_cast`——`const_cast`——使用新风格的强制

14.1 主要扩充

模板（第15章），异常（第16章），运行时类型信息（14.2节）和名字空间（第17章）经常被说成是主要扩充。之所以称它们为主要的，是因为它们对程序的组织方式产生了影响——无论是看作扩充，还是看作C++的集成特征。因为从根本上说，创造C++就是为了提供组织程序的新方式，而不是为表述传统设计提供另一种更方便的形式。所以，能在这方面起作用的就算是主要特征。

次要特征之所以被认为是次要的，也是因为它们并不影响程序的整体结构，不影响设计。说它们次要，并不是因为在手册里定义它们用的行数不多，或者只需要不多几行编译程序代码就能够实现它们。实际上，一些主要特征比某些次要特征还容易描述，或者是更容易实现。

很自然，并不是每个特征都能很好地纳入这种简单的主要/次要分类。例如，嵌套函数既可以看作次要特征也可以看作主要特征，这要看你对于它们在表述迭代中重要性如何认识。不管怎样，我这些年在试着尽可能减少次要扩充的同时，一直在几个主要扩充方面努力工作。非常奇怪的是，人们感兴趣的程度和公众争论的量却似乎常常与特征的重要性成反比。个中缘由也很容易理解：与主要特征相比，人们对次要特征更容易有一种稳固的观点；次要特征常常直接顺应了当前的形势，而主要特征——按照定义就不是这样的。

由于支持库和从部分独立的部分出发组合软件都是C++的最关键目标，主要特征全都与这些东西有关系：模板、异常处理、运行时的类型识别、以及名字空间。当然，如果按我的观点，模板和异常处理应该是在Release 1.0以前就提供的东西（2.9.2节和3.15节）。运行时类型识别甚至在C++的第一个草稿里就考虑过（3.5节），但后来又推迟了，是希望能证明它并不必要。名字空间是仅有的一个超出了C++初始概念的主要扩充，然而甚至它也是对我在C++第一个版本中就想解决，但却没有成功的一个问题的回应（3.12节）。

14.2 运行时类型信息

关于在运行时确定对象类型机制的讨论，在许多方面与关于多重继承（12.6节）的讨论很

类似。多重继承被看作对原始C++定义的第一个主要扩充。运行时类型信息（run-time type information）通常被记作RTTI，是在标准化过程委托的和ARM里发表的特征之外的第一个主要扩充。

C++又直接支持了一种新的程序设计风格。同样又有些人

- 声称这种支持是不必要的。
- 声称这种新风格从本质上说就是罪恶的（“与C++的精神相悖”）。
- 指出它太昂贵。
- 认为它太复杂、太混乱。
- 将它看成是又一次新特征雪崩的开始。

此外，RTTI还吸引来许多对C/C++强制机制的一般性批评。例如，许多人不喜欢（老风格的）强制可用于越对对私用基类的访问控制，能够强制去掉const。这些批评都很有根据，很重要。在14.3节将讨论它们。

又一次的，我为新特征辩护还是站在这样的基础上：它对于一些人是有用的，而对不使用它的另一些人则是无害的；如果我们不直接支持它，人们也会去模拟它。还有，就是它应该很容易实现。为支持最后这个断言，我用了两个上午做出了一个试验性实现。这就使RTTI至少比异常和模板简单两个数量级，比多重继承简单一个数量级。

把在运行时确定对象类型的机制加进C++，其原始动力来自Dmitry Lenkov [Lenkov, 1991]。Dmitry的想法转而来自一些重要C++库的经验，例如Interviews [Linton, 1987]，NIH库 [Gorlen, 1990]，以及ET++ [Weinand, 1988]。档案（dossier）机制 [Interrante, 1990]也是那时可以考查的东西。

由库提供的RTTI机制是互不兼容的，因此也就成为了同时使用多个库的一种障碍。所有这些还要求基础类的设计师们极富远见。因此，确实需要有一种语言支持机制。

我逐渐涉足到这个机制的细节设计中，和Dmitry一起作为提交给ANSI/ISO委员会的建议的合作者，同时也作为在委员会里精炼这个建议的负责人 [Stroustrup, 1992]。这个建议第一次提交给委员会是在1991年7月的伦敦会议上，在1993年3月俄勒冈的Portland会议上它被接受了。

运行时类型信息机制包括3个主要部分：

- 一个运算符dynamic_cast，给它一个指向某对象的基类指针，它能得到一个到这个对象的派生类指针。只有在被指对象确实属于所指明的派生类时，运算符dynamic_cast才给出这个指针，否则就返回0。
 - 一个运算符typeid，它对一个给定的基类指针识别出被指对象的确切类型。
 - 一个结构type_info，作为与有关类型的更多运行时类型信息的挂接点（hook）。
- 为了节约篇幅，下面有关RTTI的讨论几乎完全限制在指针方面。

14.2.1 问题

假定某个库里提供了类dialog_box，它的界面都用dialog_box的方式表述。而我将同时使用dialog_box和我自己的dbox_w_str：

```
class dialog_box : public window { // library class
```

```

    // ...
public:
    virtual int ask();
    // ...
};

class dbox_w_str : public dialog_box { // my class
    // ...
public:
    int ask();
    virtual char* get_string();
    // ...
};

```

在这种情况下，当系统/库传递给我一个到dialog_box的指针时，我怎么才能知道它是不是一个我的dbox_w_str？

请注意，我无法去修改库使它能知道我的dbox_w_str类。即使能我也不应该这样做，因为这样做之后我就要担心dbox_w_str类能不能在库的新版本里使用，还要担心我可能将错误引入“标准的”库里。

14.2.2 dynamic_cast运算符

一个朴素的解决办法就是找出被指对象的类型，而后将它与我的dbox_w_str类比较：

```

void my_fct(dialog_box* bp)
{
    if (typeid(*bp) == typeid(dbox_w_str)) { // naive

        dbox_w_str* dbp = (dbox_w_str*)bp;

        // use dbp
    }
    else {

        // treat *bp as a ``plain'' dialog box
    }
}

```

给了类型的名字作为运算对象，typeid()运算符将返回一个能够标识它的对象。如果给它一个表达式运算对象，typeid()运算符将返回一个对象，它标识了这个表达式所表示的对象的类型。特别是运算 typeid(*dp) 将返回一个对象，使程序员能够提出有关被dp所指对象的类型方面的问题。在上面的情况中，我们问的是这个类型是否与dbox_w_str的类型相同。

这是能提出的最简单的问题，但它也常常不是正确的问题。这里提问的原因是想知道能否安全地使用某个派生类的一些细节。而为了使用它，我们需要获得一个指向派生类的指针。在上面例子里，我们在检测之后的程序行里用了一个强制。进一步说，我们常常并不真正关心被指对象的确凿类型，而只是关心我们能否安全地做这个强制。这个问题可以直接通过dynamic_cast运算符提出：

```
void my_fct(dialog_box* bp)
```

```

{
    if (dbox_w_str* dbp = dynamic_cast<dbox_w_str*>(bp)) {
        // use dbp
    }
    else {
        // treat *pb as a ``plain'' dialog box
    }
}

```

如果*p确实是一个T或者是由T派生的类的对象，这里的dynamic_cast<T*>(p)运算符就将其运算对象p转换到所需要的T*类型；否则dynamic_cast<T*>(p)的值就是0。

将测试与强制合并为一个操作有许多优点：

- 动态强制使测试和强制之间不会出现匹配错误的情况。
- 通过利用在对象里可用的类型信息，我们可能将它转换到在这个强制的作用域里看不到完整定义的某个类型。
- 通过利用在对象里可用的类型信息，我们经常可以从一个虚的基类强制到一个派生类去（14.2.2节）。
- 静态强制不能对所有这些情况都给出正确结果（14.3.2节）。

对于我遇到过的所有与强制有关的需求，dynamic_cast能用于其中的大多数情况。我把dynamic_cast看作是RTTI机制中最重要的东西，也是用户最应该关注的结构。

dynamic_cast运算符还可用于把运算对象强制到某个引用类型。如果到引用的强制操作失败，那就会抛出一个bad_cast异常。例如：

```

void my_fct(dialog_box& b)
{
    dbox_w_str& db = dynamic_cast<dbox_w_str&>(b);
    // use db
}

```

仅在我有一个关于被检查引用类型的假设，并认为如果这个假设错误那么就是失败的时候，我才使用一个引用强制。如果情况并不是这样，我希望在多种可能性中做出选择，那么就用一个指针强制并测试其结果。

我已经无法准确回忆起，从什么时候起我就认定运行时检查的强制是处理运行时类型最好的方式，这将使语言直接支持运行时的类型检查变成了一种必需品。这个想法的第一次出现是在1984或1985年期间，当我访问Xerox PARC时有人向我建议的。该建议提出让常规的强制去做检查。正如14.2.2节中论述的，那种方式在开销和兼容性方面存在一些问题。当时我也觉得某种形式的强制能有助于尽量减少错误使用，像Simula的INSPECT那样一种基于类型的开关机制就很有吸引力。

1. 语法

dynamic_cast运算符看起来应该是什么样？有关的讨论既反映了纯粹语法上的考虑，也反映了对转换的性质方面的考虑。

强制是C++里最容易引起错误的功能之一，它们在语法上也是最难看的东西。很自然，我也考虑过是否有可能：

- 1) 删除强制。
- 2) 设法使强制成为安全的。
- 3) 为强制提供一种语法形式, 使正在使用一种不安全运算符的情况更容易被看清楚。
- 4) 为强制提供一些替代品, 不鼓励使用强制。

简单地看, 要说dynamic_cast (动态强制) 反映出3)和4)的组合还能说得通, 另一方面, 它与1)和2)就没有什么关系。

考虑第1)点, 我们注意到, 任何支持系统程序设计的语言都不可能完全清除强制, 即使是为了有效地支持数值计算, 往往也需要某种形式的类型转换。这样, 目标应该是尽可能减少强制的使用, 并尽可能将这种东西的行为弄得更好一些。从这个愿望出发, Dmitry和我制定了一个方案, 把动态强制和静态强制用老的语法形式统一起来。这看起来是个好主意, 但经过更细致的检查, 却发现了几个问题:

- 1) 动态强制和常规不加检查的强制从本质上讲就是不同的运算。动态强制需要查看对象的内部, 以便生成相应的结果; 它也可能失败, 这时就在运行中给出信息, 指明出现了失败。常规强制的执行则是一种完全由所涉及的类型决定的运算, 根本不依赖于有关对象的值 (除了偶尔做空指针检查之外)。常规强制绝不会失败, 它只是简单地产生一个新值。为这两种动态和静态强制使用同样的语法只能引起混乱, 使人弄不清一个强制到底做的是什么。
- 2) 如果动态强制在语法上不能清楚辨认, 那么就无法简单地找到它们 (去grep^Θ 它们, 按照UNIX的说话方式)。
- 3) 如果动态强制在语法上不能清楚地辨认, 编译程序就无法拒绝不适当使用动态强制的情况, 它必然会简单地根据有关类型去执行能做的强制。如果能够辨认出它们, 企图对一个不支持运行时检查的对象做动态强制就将导致一个错误。
- 4) 如果在一切可能的地方都做运行时检查, 使用常规强制的程序就可能改变意义。这方面的例子如强制到无定义的类, 或是在多重继承的层次结构中做强制 (14.3.2节)。我们无法使自己确信, 说这种改变绝不会改变所有合理程序的意义。
- 5) 检查可能惹出很大的代价, 即使是对那些已经仔细检查过的老程序, 因为完全可能出现强制改用了其他意义的情况。
- 6) 有人建议的“关闭检查”方式 (对于转换到或转换出*void) 不可能完全可靠, 因为在某些情况下意义也会改变。这类情况很可能散布在许多地方, 因此就需要理解代码, 也将使关闭检查变成一种手工操作, 极易出错。我们反对给任何可能给程序增加不加检查强制的技术。
- 7) 使某些强制变“安全”也能使强制受到更多尊重。然而更长期的目标还是减少强制的使用 (包括动态强制)。

经过许多讨论之后, 我们找到了下面的说法: “在我们理想的语言里允许出现多于一种类型转换的记法吗?”对于一个希望能用语法形式去区分本质上不同的操作的语言, 回答应该是允许。于是我们放弃了“劫持”老语法形式的企图。

我们认为, 如果能不顾及强制的原有语法形式, 下面这样的形式比较令人满意:

^Θ grep是UNIX中最常用的一种工具, 常被用于在文本文件 (例如源程序文件) 中检索包含某个字符串或字符串模式的行。由此UNIX的人们就把在文件里检索东西直称为grep……。——译者注

```
Checked<T*>(p); // run-time checked conversion of p to a T*
Unchecked<T*>(p); // unchecked conversion of p to a T*
```

这将最终使所有转换都变得很明确，而且能清除由于传统强制在C和C++程序里不易辨认而引起的问题。这也给所有强制提供了一种共同的语法模式，它基于模板（第15章）中有关类型的记法。由这种思维方式引出的强制的另一种语法形式见14.3节。

并不是每个人都喜欢模板的语法，当然，也不是每个喜欢模板语法的人都乐于用它作为强制运算符。因此我们又讨论和试验了另外一些形式。

记法`(?T*)p`流行过一段时间，因为它直接模仿了传统的语法形式`(T*)p`。另一些人不喜欢它，也是因为同样的理由。许多人还认为`(?T*)p`“太像密码”。不对，我发现了最关键的失误之处。在使用`(?T*)p`时，最常见的错误将是丢掉其中的`?`。这样做实际上就会使一个相对安全的、需要经过检查的转换变成了一个完全不同的、不安全的、根本不检查的操作。例如：

```
if (dbox_w_string* p = (dbox_w_string*)q) // dynamic cast
{
    // *q is a dbox_w_string
}
```

真糟糕！忘记了问号就使这里的注释变成了一句谎话，这可能会变得很常见，太叫人不习惯了。请注意，我们不能通过用grep找出所有老风格强制的方式来防止出现这类错误，因为极强的C语言背景将使我们很容易犯这种错误，而在读代码时又会忽略它。

在考虑选择的各种形式中，记法

```
(virtual T*)p
```

最引人注目。它比较容易由人和工具辨认，单词`virtual`指出了与带有虚函数的类的逻辑联系（多态类型），这样普通的语法模式也很像传统强制的形式。但许多人还是认为它“太像密码”。它还引起了那些根本不喜欢强制的老语法形式的人的敌意。从个人角度说，我有些同意这些批评，觉得`dynamic_cast`的语法形式能更好地放进C++（许多对模板有较多使用经验的人们也这样认为）。我也认识到另一个优势：`dynamic_cast`提供了一种清晰的语法框架，使它有可能最终变成老强制的一种替代形式（14.3节）。

2. 何时能使用动态强制

引进了运行时的类型识别，实际上就把对象分成了两种：

1) 一种包含了与之关联的类型信息，因此它们的类型（几乎）总能被确定，与上下文没有关系。

2) 另一种不能。

为什么？我们不可能把可以在运行时确定对象类型的负担强加到各种内部类型（例如`int`和`double`）上，同时又不在运行时间、空间、布局兼容性等方面付出无法接受的代价。类似的论断还适用于简单的类对象和C语言风格的`struct`。因此，从实现的观点看，第一条可以接受的分界线是在带虚函数的类对象与没有虚函数的类之间。前者很容易提供类型信息，后者就不容易。

进一步说，带虚函数的类也常常被称为是多态类，而只有多态类才能通过基类安全地进行操作。对于“安全”，我是想说，语言提供了一种保证，保证这些对象能够只按照它们的定

义类型被使用。当然，个别程序员也可以在一些特殊情况下做出一些演示，说明对一些非多态类的操作也不违反类型系统。

从程序员的观点看，只对多态类型提供运行时类型识别机制看起来也很自然：它们恰好就是C++所支持的能够通过基类去操作的那些东西。对非多态类提供RTTI，实际上就是为增加类型域的程序设计提供支持。当然，程序设计语言不应该反对这种程序设计，但是我也完全看不出有什么必要为迎合这种东西而使语言进一步复杂化。

经验说明，只为多态类型提供RTTI也是可以接受的。但不管怎样，人们还是可能在哪些对象是多态的问题上感到困惑，因此就弄不清楚能不能使用动态强制。幸运的是，如果程序员的猜测不对，编译程序总能够捕捉到这些错误。我找了很长时间，也很努力，想找一种可以接受的明显方式去说“这个类型支持RTTI（无论它有没有虚函数）”，但却始终无法找到一个值得花精力去引进来的东西。

3. 从虚基出发的强制

引进了dynamic_cast运算符，也使我们有了一种方式去绕过一个老问题。使用常规强制不可能实现从一个虚基类到它的某个派生类的强制。存在这种限制的原因是在对象里没有足够的信息，无法实现从一个虚基类到它的一个派生类的强制，见12.4.1节。

但是现在，在为提供运行时类型识别所需要的信息里，已经包含了实现从虚基类出发的动态强制所需要的信息。这样，无法从虚基类出发做强制的限制并不适用于从多态虚基类出发的动态强制：

```
class B { /* ... */ virtual void f(); };
class V { /* ... */ virtual void g(); };
class X { /* no virtual functions */ };

class D: public B, public virtual V, public virtual X {
    // ...
};

void g(D& d)
{
    B* pb = &d;
    D* p1 = (D*)pb; // ok, unchecked
    D* p2 = dynamic_cast<D*>(pb); // ok, run-time checked

    V* pv = &d;
    D* p3 = (D*)pv; // error: cannot cast from virtual base
    D* p4 = dynamic_cast<D*>(pv); // ok, run-time checked

    X* px = &d;
    D* p5 = (D*)px; // error: cannot cast from virtual base
    D* p6 = dynamic_cast<D*>(px); // error: can't cast from
                                    // non-polymorphic type
}
```

当然，只有在能够无歧义地确定派生类的情况下，才能执行这种强制。

14.2.3 RTTI的使用和误用

只有必须用的时候，才应该明确地使用运行时类型信息。静态（编译时）检查更安全，

带来的开销更少，而且——在能用的地方——将导致结构更好的程序。例如，RTTI可以被用于写经过无聊伪装的开关语句：

```
// misuse of run-time type information:

void rotate(const Shape& r)
{
    if (typeid(r) == typeid(Circle)) {
        // do nothing
    }
    else if (typeid(r) == typeid(Triangle)) {
        // rotate triangle
    }
    else if (typeid(r) == typeid(Square)) {
        // rotate square
    }
    // ...
}
```

我听到过有人把这种风格描述为提供了“C语言的优美语法与Smalltalk运行效率的组合”[⊖]，而这种说法还是太客气了。这些代码的真正问题在于它无法正确处理由这里正确描述的类进一步派生出新的类：只要有新类加进程序，这些代码就必须修改。

这种代码通常都可以通过虚函数避免。正是根据在Simula里写这类形式代码所得到的经验，我才在刚开始设计C++时把运行时类型识别扔到了一边（3.5节）。

对于由C、Pascal、Modula-2、Ada等语言训练出来的许多人来说，将程序组织为一集丑陋的开关语句的形式，几乎是无法避免的情况。请注意，虽然标准化委员会赞成在C++里提供RTTI，我们可没有通过用类型开关语句的形式（像Simula的INSPECT语句）来支持这种概念。我始终不认为以类型开关语句的方式去组织程序应该得到直接支持。真正应该以这种方式组织程序也是有的，但是比大部分程序员在一开始时认为的要少得多——而到了程序员们再次考虑时，需要做的重新组织通常都涉及到太多的工作。

正确使用RTTI的许多例子大都出自这类地方：在其中服务性的代码用一个类表示，用户又希望通过派生为它增加新功能。14.2.1节的dialog_box就是这样的一个例子。如果用户希望并且能够直接修改库类的定义，例如修改dialog_box的定义，那么就可以避免使用RTTI；如果行不通，就必须用这种机制了。即便用户希望去修改基类，这种修改也可能有它自己的问题。例如，修改时可能需要为get_string()这类函数引进哑的实现，而且还是把这种虚函数引进到它们根本没用也没有意义的一些类里。[2nd, 13.13.6节]里在“肥大的界面”的名目下讨论了这个问题的一些细节。有关用RTTI实现一个简单的I/O对象库的例子可以在14.2.7节中找到。

那些在依靠动态类型检查的语言（如Smalltalk）方面有深厚基础的人常常会想去使用RTTI，同时使用过于一般的类型。例如：

```
// misuse of run-time type information:

class Object { /* ... */ };
```

[⊖] C语言的语法不优美，Smalltalk语言的运行效率不太高。这个话是反话，是讥讽。——译者注

```

class Container : public Object {
public:
    void put(Object*);
    Object* get();
    // ...
};

class Ship : public Object { /* ... */ };

Ship* f(Ship* ps, Container* c)
{
    c->put(ps);
    // ...
    Object* p = c->get();
    if (Ship* q = dynamic_cast<Ship*>(p)) // run-time check
        return q;

    // do something else (typically, error handling)
}

```

在这里的类Object就是一个完全没有必要实现的人为现象。它过于一般，因为它并不对应于实现领域中的任何抽象。它还迫使程序员在过低的抽象层次上操作。要解决这类问题，更好的方法是使用容器模板，在其中只保存某一类指针：

```

template<class T> class Container {
public:
    void put(T*);
    T* get();
    // ...
};

Ship* f(Ship* ps, Container<Ship*>* c)
{
    c->put(ps);
    // ...
    return c->get();
}

```

通过与虚函数的使用相配合，这种技术可以处理绝大部分情况。

14.2.4 为什么提供一个“危险特征”

那么，既然我能深信不疑地断言RTTI的误用，为什么我还要设计这个机制，并为它被接受而努力工作呢？

要获得好的程序，需要经过好的教育、好的设计、充分的测试等等，不可能只是通过提供一些在想象中只能支持“按正确方式”使用的语言机制。每个有用的特征都可以被误用，所以，问题并不在于一种特征能不能被误用（当然能），或者说是否将被误用（当然会）。真正的问题是，对一种特征的正确应用是否充分重要，值得为提供它去花费那些精力；用语言的其他机制来模拟这种特征是不是容易处理；通过正确的教育，能不能把对它的错误使用控制在合理的范围之内。

※ 在考虑RTTI一段时间之后，我逐渐确信我们面对的确实是一个经典的标准问题：

- 大部分主要的库都提供了某种RTTI特征。
- 提供这种特征的大部分形式都要求用户做一些非常重要的、但是也很容易出错的配合，以便使这种特征能够正确工作。
- 各种东西都以互不兼容的方式提供它。
- 大部分都是以某种不够一般的形式提供它。
- 大部分都以某种“好特征”的方式提供，希望用户去用它们；而不是作为一种危险的特征，只是希望作为一种最后的退路。
- 看来在每个主要的库里都（并且仅仅）存在几种情况，对于这些情况，RTTI是极其关键的。也就是说，如果没有RTTI，这个库就可能无法提供某种功能，或者只能通过给用户和实现者强加上一些明显负担的方式来提供这些功能。

通过提供标准的RTTI，我们就能克服在使用不同来源的库时所遇到的一个障碍（8.2.2节）。我们可能给RTTI的使用提供一种统一的观点，设法尽量将它做得更安全些，并通过提供警告信息的方式防止对它的误用。

最后，在C++设计中有一条指导原则，那就是，无论做什么事情都必须相信程序员。与可能出现什么样的错误相比，更重要得多的是能做出什么好事情。C++程序员总被看作是成年人，只需要最少的“看护”。

当然，并不是每个人都能被说服。有些人，特别是Jim Waldo，就激烈地论证说需要RTTI的情况极少出现，而作为误用RTTI之根源的错误概念则散布广泛，RTTI的最终影响必将成为在C++里很有害的东西。只有时间能够给出确切的证据。错误使用的最大危险可能来自某些程序员，他们自认为非常专业，根本看不到在使用C++之前还需要去咨询一下C++的教科书（7.2节）。

14.2.5 typeid()运算符

我本来希望仅有dynamic_cast运算符就能满足所有的常见需要了。如果真是那样，也就没有必要再为用户提供其他RTTI机制了。但是，大部分与我讨论这个问题的人都不同意这种看法，他们指出了另外两方面的需求：

- 1) 可能需要确定一个对象的确切类型。也就是说，告诉说这个对象就是X类的对象，而不是只说，它是X类的或者某个由X类派生的类的对象。dynamic_cast做的是后一件事情。
- 2) 以一个对象的确切类型作为过渡，得到描述该类型其他性质的信息。

找出一个对象的确切类型有时被称作类型标识，因此我给这个运算符起名为typeid。

人们希望知道对象的确切类型，通常是他们因为想对这个对象的整体执行某种标准服务。在理想的情况下，这种服务是通过虚函数提供的，这时就不必知道对象的确切类型了；但是，如果因为某些原因没有这样的函数可用，那么就必须找到对象的确切类型，而后再执行有关的操作。人们已经按照这种方式设计了I/O对象系统和数据库。在这些情况下，根本无法假定对每种对象的操作能够有一个公共的界面，所以只能另辟蹊径，利用对象的确切类型就变得不可避免了。另一种更简单的使用是想取得类的名字，以便产生某些诊断输出。

```
cout << typeid(*p).name();
```

typeid()运算符要以明显的方式使用，以便在运行时取得有关类型的信息，它是一个内部

运算符。如果 typeid() 是个函数，那么它的声明大概是下面的样子：

```
class type_info;
const type_info& typeid(type-name); // pseudo declaration
const type_info& typeid(expression); // pseudo declaration
```

也就是说， typeid() 返回到某个未知类型 type_info 的引用^Θ。给它一个 type_name (类型名) 作为参数， typeid() 返回到一个 type_info 的引用，该 type_info 表示这个 type_name。给定一个 expression (表达式) 作为参数， typeid() 返回到一个 type_info 的引用，这个 type_info 表示 expression 所指称的对象的类型。

让 typeid() 返回一个 type_info 引用 (而不是返回对应的指针)，是因为我们不想允许在 typeid() 的结果上使用通常的指针运算，如 == 和 ++ 等。例如，我们不清楚是否每个实现都能保证类型识别对象的唯一性，这就意味着不能将 typeid() 之间的比较简单地定义为到 type_info 的指针之间的比较。让 typeid() 返回一个 type_info&，这时定义 ==，就能同时处理一个类型对应出现多个 type_info 的情况了。

1. type_info 类

type_info 类在标准头文件 <type_info.h> 里定义，如果想使用 typeid() 的结果，就需要包含这个头文件。type_info 类的确切定义是与实现有关的，但它应该是一个多态的类型，提供了比较和另外一个操作，该操作返回所表示的类型名：

```
class type_info {
    // implementation-dependent representation

private:
    type_info(const type_info&);           // users can't
    type_info& operator=(const type_info&); // copy type_info

public:
    virtual ~type_info();                  // is polymorphic

    int operator==(const type_info&) const; // can be compared
    int operator!=(const type_info&) const;
    int before(const type_info&) const;     // ordering

    const char* name() const;              // name of type
};
```

还可以提供更详细的信息，供外面访问。有关情况在下面描述。但无论如何，由于不同的人对“更详细信息”的需要差异很大，也因为另一些人特别要求最小的空间开销，所以 type_info 提供的服务被有意地规定为最小的。

函数 before() 是为了使 type_info 信息能够排序，以便能通过散列表等方式访问它们。由 before() 定义的顺序关系和继承关系之间没有任何联系 (14.2.8 节)。进一步说，对不同的程序或者同一个程序的不同运行，我们都不能保证 before() 能产生同样的结果。在这个方面， before() 与取地址运算类似。

2. 扩充的类型信息

也有些时候，知道了对象的确切类型只不过是作为获取和使用有关该类型的更细节信息

^Θ 标准化委员会还在讨论标准库类的命名规则。我取的是自己认为最可能成为这个讨论的结果的名字。

的第一步。

现在考虑在运行时一个实现或工具怎样才能将有关类型的信息提供给用户使用。比如说我们有一个工具，它生成一个名为My_type_info的对象表。将这种东西表达给用户的最好方式是提供一个类型名与这种表的关联数组（映射，字典）。如果要对某个类型取得这样一个成员表，用户可能写：

```
#include <type_info.h>

extern Map<My_type_info, const char*> my_type_table;

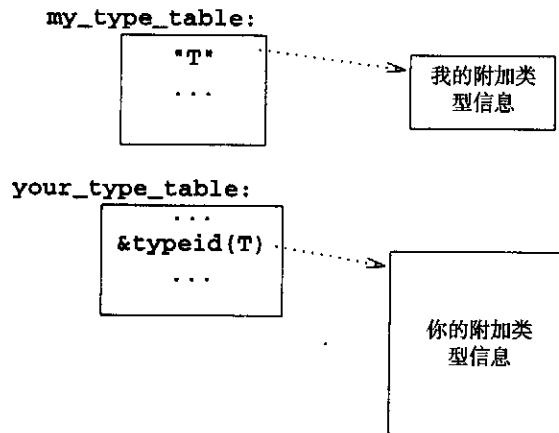
void f(B* p)
{
    My_type_info& mi = my_type_table[typeid(*p).name()];
    // use mi
}
```

另外一些人可能更喜欢直接用typeid作为下标，去取对应的表，而不是要求用户去使用类型的name()串：

```
extern Map<Your_type_info, type_info*> your_type_table;

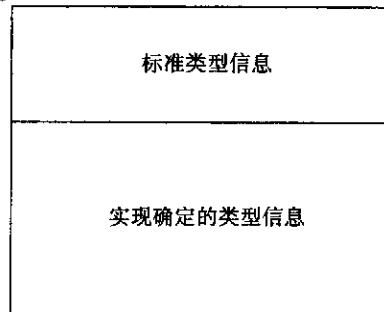
void g(B* p)
{
    Your_type_info& yi = your_type_table[&typeid(*p)];
    // use yi
}
```

采用这类方式将typeid与有关信息建立关联，就能允许不同的人或工具给类型关联不同的信息，而又不会互相干扰：



这是最重要的，因为某个东西携带的信息能满足所有用户的情况是不大可能出现的。特别是任何能满足大部分用户的信息集合都会非常大，对于那些只需要最少运行时类型信息的用户而言，这种开销是绝不能接受的。

某个实现可以选择性地提供一些由实现确定的附加的类型信息，这种特定系统提供的扩充的类型信息能够通过一个关联数组访问，恰如用户所提供的扩充信息一样。也可以换另一种方式，通过一个由type_info派生的类Extended_type_info去给出扩充的类型信息：

type_info:

这样就可以通过dynamic_cast确定能否使用某一种扩充的类型信息：

```
#include <type_info.h>

typedef Extended_type_info Eti;

void f(Sometype* p)
{
    if (Eti* p = dynamic_cast<Eti*>(&typeid(*p))) {
        // ...
    }
}
```

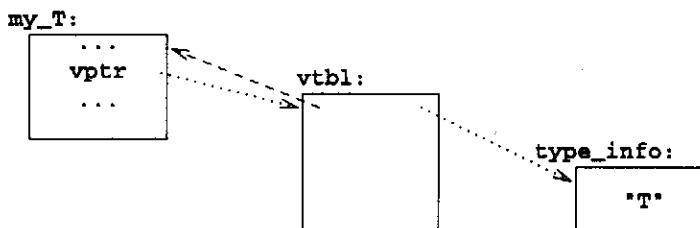
工具或实现能为用户提供什么样的“扩充”信息呢？简单地说，这其中可以包括编译程序能提供的，某些程序在运行时根据需要产生的任何信息。例如：

- 为对象I/O或者排除程序错误所用的对象布局信息。
- 指向建立或复制对象的函数的指针。
- 函数与它们的符号名字的关联表，以便在解释性代码中调用。
- 给定类型的所有对象的列表。
- 对成员函数源代码的引用。
- 类的在线文档。

这种东西需要通过库提供，也可能是通过标准库。产生这种情况的原因是存在的需求太多，潜在的由具体实现规定的细节太多，为支持语言本身的所有使用而需要的信息也太多。还有，在这之中，有些使用可能推翻语言提供的静态检查。另一些将给运行时间或空间添加很大的代价，我觉得都不适合作为语言特征。

14.2.6 对象布局模型

下面是一个对象的可能存储布局形式，该对象带有虚函数表和类型信息对象：



这里的短线箭头表示一个偏移量，有了它，只要有一个指向多态子对象的指针，就能够确定整个对象的开始位置了。这等价于虚函数实现中所使用的偏移量 (delta, 12.4节)。

对每个有虚函数的类型，将生成一个type_info类型的对象，这些对象不必是唯一的。当然，好的实现应当尽可能唯一地生成type_info对象，而且只对那些在实际中真正用到运行时类型信息的类型生成对应的type_info对象。一种最容易实现的方法就是把一个类的type_info对象放在它的vtbl之后。

Cfront的基本实现以及那些直接借用了Cfront虚函数表布局方式的实现都可以升级为支持RTTI，甚至不需要重新编译老的代码。其中的原因是，我在实现Release 2.0时已经考虑了提供RTTI的事情，在每个vtbl的开始处留下了两个空字，以便用于这种扩充。那时我没有加进RTTI，因为还不能确定是否真正需要它，或者假定确实需要，但也还没有确定到底应该将什么功能展现给用户。作为试验我实现了一个简单的版本，在其中每个有虚函数的类的各个对象都被做成能打印类名字。做了这些之后，我感到满意了，知道在RTTI变为必需品时应该如何把它加进去——而后就删除了这个特征。

14.2.7 一个例子：简单的I/O对象

让我勾勒出一个草图，说明用户可能如何与一个简单的对象I/O系统一起使用RTTI，并描述一个这样的对象I/O系统的可能实现方式。用户希望能通过流直接读对象，确定它们具有预期的类型，而后使用它们。例如：

```
void user()
{
    // open file assumed to hold shapes, and
    // attach ss as an istream for that file
    // ...

    io_obj* p = get_obj(ss); // read object from stream

    if (Shape* sp = dynamic_cast<Shape*>(p)) {
        sp->draw(); // use the Shape
        // ...
    }
    else {
        // oops: non-shape in Shape file
    }
}
```

函数user()仅仅根据抽象类Shape处理各种形状 (shape)，因此它可以使用任何形状。在这里就必须使用dynamic_cast，因为对象I/O系统应该能处理许多其他类型的对象，而用户可能偶然地打开了某个文件，其中包含的确实是属于某个类的完好的对象，但是用户可能从来都没听说过这些类。

这个I/O系统假定，所有对象的读写都通过由io_obj派生的某个类进行。io_obj必须是一个多态类，以便我们能使用dynamic_cast。例如：

```
class io_obj { // polymorphic
    virtual io_obj* clone();
};
```

对象I/O系统中最关键的函数就是get_obj()，它从流中读入数据，并基于这些数据把对象建立起来。现在假定，由输入流获得的表示对象的数据总以一个标识对象类的字符串作为前缀。这样，get_obj()的工作就是读入这个串，而后调用一个适当的函数，它能够正确地建立和初始化这个类的对象。例如：

```

typedef io_obj* (*PF)(istream&);

Map<String, PF> io_map; // maps strings to creation functions

io_obj* get_obj(istream& s)
{
    String str;
    if (get_word(s, str) == 0) // read initial word into str
        throw no_class;

    PF f = io_map[str]; // lookup 'str' to get function
    if (f == 0) throw unknown_class; // no match for 'str'
    io_obj* p = f(s); // construct object from stream
    if (debug) cout << typeid(*p).name() << '\n';
}

```

这里被称为io_map的Map是一个关联数组，其中保存着一些名字串和对应函数的对偶，这些函数能建立起具有相应名字的类的对象。Map类型是在任何语言里都最有用最有效的一种数据结构。在C++语言里，这个想法的第一个广泛使用的实现是Andrew Koenig写的 [Koenig, 1988]；也见 [2nd, 8.8节]。

请注意，这里的typeid()用于程序排错。在这个特定的设计里，这是实现中唯一真正用到RTTI的地方。

我们当然可以像平常一样定义Shape类，除了应该令它从io_obj派生之外，同时也是函数user()所需要的：

```

class Shape : public io_obj {
    // ...
};

```

如果能够不加修改地使用前面已经定义的Shape类层次，那当然是更有意思的（在许多情况下也是更实际的）：

```

class iocircle : public Circle, public io_obj {
public:
    iocircle* clone() // override io_obj::clone()
    { return new iocircle(*this); }

    iocircle(istream&); // initialize from input stream

    static iocircle* new_circle(istream& s)
    {
        return new iocircle(s);
    }
    // ...
};

```

这里的*iocircle(istream&)*建构函数用它从自己的*istream*参数得到的数据去初始化一个对象。*new_circle*就是应该被放进*io_map*里，使对象I/O系统能够知道这个类的函数。例如写：

```
io_map["iocircle"] = &iocircle::new_circle;
```

其他形状所对应的东西都按同样的方式构造：

```
class iotriangle : public Triangle, public io_obj {
    // ...
};
```

如果建立对象I/O的框架变得太冗长乏味，那么就可以用一个模板：

```
template<class T>
class io : public T, public io_obj {
public:
    io* clone() // override io_obj::clone()
    { return new io(*this); }

    io(istream&); // initialize from input stream

    static io* new_io(istream& s)
    {
        return new io(s);
    }
    // ...
};
```

有了这些，我们就能如下地定义*iocircle*了：

```
typedef io<Circle> iocircle;
```

我们当然还需要明确地定义*io<Circle>::io(istream&)*，因为它必须知道*Circle*的实现细节。

这个简单的对象I/O系统可能没有包含人们想要的所有东西，但它几乎可以放进一页纸里，其中许多地方使用了各种关键机制。一般地说，这种技术可以用到基于用户提供的字符串调用函数的任何地方。

14.2.8 考虑过的其他选择

这里提供的RTTI采用了一种“洋葱头式的设计”。当你剥去这个机制的一层皮后，你将发现更强有力的功能——如果用得不好也定能叫你痛哭流涕。

这里描述的RTTI机制的基本概念是最容易用于程序设计的，最少可能依赖于具体实现。当然，我们还是应该尽量减少RTTI的使用：

- 1) 最可取的是根本不用运行时类型信息机制，完全依靠静态的（编译时的）类型检查。
- 2) 如果这样做不行，我们最好是只用动态强制。在这种情况下我们甚至不必知道对象的类型，也不必包含任何与RTTI有关的头文件。
- 3) 如果必需的话，我们可以做*typeid()*的比较。但是在这样做时我们至少需要知道所涉及的某些类型的名字。这里假定“常规的用户”不需要进一步检查运行时类型信息。
- 4) 最后，如果确实需要知道关于一个类型的更多信息——比如说我们要实现一个排错系

统，一个数据库系统，或者其他形式的对象I/O系统——那么我们就需要使用定义在 typeid上的操作，以获得更详细的信息了。

这里采用的方式是提供一系列功能，逐步深入地去接触类的运行时特性。这种方式正好与采用元对象的方式相反，在那里，只给出了一个关于类的运行时类型特性的唯一的标准观点。C++的方式是鼓励尽可能地依靠（更安全更可靠的）静态类型系统，要求用户付出的最小代价也更小一些（无论在时间方面还是可理解性方面），提供的功能也更具一般性。因为能够对一个类提供多种不同观点，可能提供更详细的信息。

这种“洋葱头途径”的另一些替代方式也经过了认真考虑。

1. 元对象

上述洋葱头途径与Smalltalk和CLOS所采用的方式截然不同。人们也反复将这些系统所用的途径以建议的方式提给C++。在这些系统里，`type_info`被一种“元对象”取代，在运行中它能接受请求，在对象上执行这个语言里能够提供的任何操作。从本质上说，建立这种元对象机制相当于在运行环境里嵌入了一个针对整个语言的解释器。我把这看作是对语言基本执行效率的一种严重威胁，是颠覆保护系统的一条地下通道，与静态类型检查的基本设计概念及有关文档水火不容。

反对把C++放在元对象之上，并不意味着元对象不是有用的东西。它们当然可以很有用，扩展的类型信息的概念能打开一扇门，使真正需要这种东西的人可以通过库的方式提供这种东西。当然，我确实是拒绝了给每个C++用户加上这种机制的负担，而且我也不会为一般的C++程序设计推荐这种设计和实现技术，更多的细节参见[2nd, 第12章]。

2. 类型查询运算符

对许多人来说，如果某个运算符能回答问题：“*`pd`的类型是`D`还是另一个从`D`派生的类？”那么该运算符似乎比执行强制的`dynamic_cast`更自然些，当然，当且仅当有关问题的回答是肯定的。也就是说，他们希望能写下面这样的代码：

```
void my_fct(dialog_box* bdp)
{
    if (dbp->isKindOf(dbox_w_str)) {
        dbox_w_str* dbsp = (dbox_w_str*)dbp;
        // use dbsp
    }
    else {
        // treat *dbp as a ``plain'' dialog box
    }
}
```

在这里存在着几个问题。其中最严重的是，强制可能无法给出意想的结果（见14.3.2节）。这也是一个例子，说明要从其他语言输入一个概念会是多么的困难。在Smalltalk里提供了`isKindOf`，用于类型获取，但是Smalltalk并不需要随后的强制，因此也就不会受到有关问题的损害。无论如何，引进`isKindOf`概念将导致技术上和风格上的问题。

事实上，正是有关风格的论述把问题提到我的面前，好像说明某种形式的条件强制更可取些。直到我发现了上面这种“判决性的”例子，才彻底否定了像`isKindOf`一类的类型获取运算符。将检测和类型转换分离，不但累赘，也可能出现检测与强制之间不匹配的情况。

3. 类型关系

一些人建议在type_info对象上定义<、<=等，以描述类层次中的关系。这很简单，也很可爱，但也存在类似14.2.8节所描述的显式类型比较运算符的问题。我们需要的是在任何事件中只做一次强制，因此还是只能用动态强制。

4. 多重方法

RTTI更有希望的应用是支持“多重方法”，也就是说，基于不止一个对象去选择虚函数。如果要写定义在多种对象上的二元运算处理代码，这个语言功能是一种极好的东西，参见13.8节。看起来，利用type_info对象很容易保存为建立这种功能所需要的信息。这就使多重方法更有可能成为将来的一个扩充。

我一直没提出这样的建议，因为我还不能清楚地把握这种变化的实际内涵，也不希望没在C++的环境里取得经验之前就去建议一个主要扩充。

5. 不受限的方法

有了RTTI，人们就可以支持“不受限的方法”。也就是说，能在type_info对象里为某个类保存充分多的运行时检查所需的信息，无论有关函数是否支持它。这样，人们就可以写出具有Smalltalk风格的运行时检查的函数调用。当然我并不认为需要这样做，也认为这种扩充与我鼓励高效的类型安全的程序设计的一贯想法背道而驰。动态强制允许我们采用一种检查并调用的策略：

```
if (D* pd = dynamic_cast<D*>(pb)) { // is *pb a D?
    pd->dfct(); // call D function
    // ...
}
```

而不是Smalltalk的调用并带上调用检查的策略：

```
pb->dfct(); // hope pb points to something that
              // has a dfct; handle failed calls
              // somewhere (else)
```

调用并检查的策略使我们可以做更多的静态检查（我们在编译时就知道dfct对类D是有定义的），对于绝大部分（不需要检查的）调用不增加任何负担，而且为某些超出常规的东西提供了可见的线索。

6. 带检查的初始化

我们也考虑了采用类似Beta或者Eiffel等语言所用的方式，带检查的赋值和/或初始化的问题。例如：

```
void f(B* pb)
{
    D* pd1 = pb; // error: type mismatch
    D* pd2 ?= pb; // ok, check if is *pb a D at run time

    pd1 = pb; // error: type mismatch
    pd2 ?= pb; // ok, check if is *pb a D at run time
}
```

但是我认为这里的问号（?）在实际代码里太难看清楚，也太容易出错，因为它并没有与一个检测组合在一起。此外，要这样做，有时还必须引进本来不必要的变量，另一种形式是只允

许在条件里面用`?=`，看起来很诱人：

```
void f(B* pb)
{
    D* pd1 ?= pb;           // error: unchecked
                           // conditional initialization

    if (D* pd2 ?= pb) { // ok: checked
                           // conditional initialization
        // ...
    }
}
```

但是，你可能还必须能区分一些情况，例如在哪里失败将导致抛出了异常，哪里返回的是0。还有，由于`?=`运算符没有强制的恶名声，因而也更可能鼓励错误使用。

通过在条件语句里允许声明（3.11.5节），我就能采用`dynamic_cast`来写出这种替代风格所建议的东西：

```
void f(B* pb)
{
    if (D* pd2 = dynamic_cast<D*>(pb)) { // ok: checked
        // ...
    }
}
```

14.3 强制的一种新记法

无论从语法上还是从语义上看，强制都是C和C++里最难看的特征之一。这也就导致了一种持续的努力，为强制探寻各种替代品：函数声明使参数的隐式转换成为可能（2.6节）；模板（14.2.3节）、放松对虚函数的覆盖规则（13.7节）等，每个都清除掉一些对强制的需要。在另一方面，`dynamic_cast`运算符（14.2.2节）也是针对一类特殊情况，为原有的强制提供一种替代品。这导致了一种互补的途径：试图把强制的各种在逻辑上互相分离的应用分割开，通过类似`dynamic_cast`的运算符分别支持它们：

```
static_cast<T>(e)      // reasonably well-behaved casts.
reinterpret_cast<T>(e) // casts yielding values that must
                       // be cast back to be used safely.
const_cast<T>(e)       // casting away const.
```

这一节可以看作是对老风格的强制中存在的各种问题的分析，也可以看作是一个新特征的综合。它同时有两方面的作用。这些运算符的定义主要归功于扩展工作组的努力，在那里总是可以听到很强烈的支特和反对的声音。Dag Brück, Jerry Schwarz和Andrew Koenig做出了特别有建设性的贡献。这些新的强制运算符是在1993年的San Jose会议上被接受的。

为了节约篇幅，下面的讨论将完全限于最困难的情况：指针。对于算术类型、到成员的指针、引用等情况的处理都留下作为读者的练习。

14.3.1 问题

C和C++的强制是一把长柄大锤：(T)expr从基于expr的值出发，按照某种方式产生一

个T类型的值——除了极少的例外。其中可能涉及到对expr的二进制位重新做出解释；也可能需要做某些指针算术，或者需要在类的层次中穿行；强制的结果还可能与实现有关；可能除掉了有关的const或者volatile属性；等等。一个读者很难从强制表达式中确定写程序人的真实意图。例如：

```
const X* pc = new X;
// ...
pv = (Y*)pc;
```

程序员是想获得一个到与x无关的类型的指针吗？还是想强制除掉const？或是两者？还是意在取得对x的基类Y的访问权？能把人搞糊涂的各种可能性真是没完没了。

进一步说，一个看起来无害的对声明的修改也可能默不做声地、非常奇怪地改变了表达式的意义。例如：

```
class X : public A, public B { /* ... */ };

void f(X* px)
{
    ((B*)px)->g(); // call B's g
    px->B::g(); // a more explicit, better, way
}
```

如果改变B的定义，使x不再是它的一个基类，(B*)px的意义就完全改变了，同时也没给编译程序诊断问题留下任何机会。

老的强制除了有语义问题之外，在记法上也很不幸。这个记法接近于最小化，只使用了括号——这是C语言里最过度使用的一种语法结构。因此，人们很难在程序里看清所有的强制，用grep一类的工具也很难将它们检索出来。强制的语法也是使C语言语法分析程序复杂化的一个重要原因。

总结一下老风格的强制：

- 1) 是理解方面的一个问题：它们提供了一种单一的记法，被用于多个互相之间关系并不大的操作。
- 2) 容易引起错误：几乎每种类型组合都有某种合法的解释。
- 3) 在代码里难以辨别，难以用简单的工具检索。
- 4) 使C和C++的语法复杂化。

新的强制运算符实际上是对老的强制功能做了一种分类。为了能有机会得到用户的广泛接受，它们还必须能执行老强制能做的所有操作，否则就会给老强制的继续使用提供理由。我只发现了一个例外：老风格的强制能够从一个派生类强制到它的一个私用基类。提供这个操作没有任何理由，因为它是危险而无用的。根本不应该有这样一种机制，使得能为自己获取对一个对象的完全私用表示的访问权，也不需要有这个东西。老风格强制能用于获得这种权利，去访问表示自己的私用基类的那个部分，这完全是一个不幸的历史偶然。例如：

```
class D : public A, private B {
private:
    int m;
    // ...
};
```

```

void f(D* pd) // f() is not a member or a friend of D
{
    B* pb1 = (B*)pd; // gain access to D's
                      // private base B.
                      // Yuck!
    B* pb2 = static_cast<B*>(pd); // error: can't access
                                    // private. Fine!
}

```

除了把pd按照指向未加工存储区的指针去操作, f()没有任何其他方法去得到D::m。这样,新的强制运算符封闭了访问规则里的一个漏洞,提供了更强的一致性。

新强制的长名字和类似模板的语法就是为了提醒用户,强制是一种危险的事情,并强调,在使用不同运算符时可能涉及到不同种类的危险。在我的体验中,最强烈的不满来自那些主要把C++作为C语言的一种方言而使用的人们,他们也经常需要用强制。还有那些至今还没怎么用过模板的人,他们也会觉得这种记法有些奇怪。随着他们逐渐取得对模板的经验,不喜欢这种类似模板的记法的情绪也就会逐渐消退了。

14.3.2 static_cast运算符

记法static_cast<T>(e)是想取代(T)e,用于例如从Base*到Derived*的转换。这种转换不一定总是安全的,但经常是安全的,甚至在缺乏运行时检查的情况下也是定义良好的。例如:

```

class B { /* ... */ };
class D : public B { /* ... */ };

void f(B* pb, D* pd)
{
    D* pd2 = static_cast<D*>(pb); // what we used
                                    // to call (D*)pb.

    B* pb2 = static_cast<B*>(pd); // safe conversion
    // ...
}

```

理解static_cast的一种方式就是将它看成隐含转换的显式的逆运算。除了不能处理常量性之外,只要T->S能够隐式完成,static_cast就能做S->T。这也就意味着在大部分情况下static_cast的结果可以直接使用,不必再做进一步强制,在这方面与reinterpret_cast不同(14.3.3节)。

此外,所有可以隐含地执行的转换——例如各种标准转换和用户定义的转换——也可以用static_cast调用。

与dynamic_cast不同,static_cast对pb的转换并不要求做任何运行时检查。被pb指向的对象也可能不是一个D,在这种情况下对*pd2的使用就是无定义的,完全可能造成大灾难。

与老风格的强制不同的是,在这里指针和引用的类型必须是完全的。也就是说,如果想用static_cast在一个指针和一个类型之间转换,看不到这个类型的声明将被当作是一个错误。例如:

```

class X; // X is an incomplete type
class Y; // Y is an incomplete type

void f(X* px)
{
    Y* p = (Y*)px; // allowed, dangerous
    p = static_cast<Y*>(px); // error:
                                // X and Y undefined
}

```

这清除了另一类错误。如果你需要强制到一个不完全类型，那就该用`reinterpret_cast`（14.3.3节）说清楚你并不是想在类层次上穿行；或者就该用`dynamic_cast`（14.2.2节）。

静态强制和动态强制

对于指向类的指针而言，`dynamic_cast`（动态强制）和`static_cast`（静态强制）的作用都是在类层次上穿行。当然`static_cast`完全依赖静态的信息，因此就可以欺骗它。考虑：

```

class B { /* ... */ };

class D : public B { /* ... */ };

void f(B* pb)
{
    D* pd1 = dynamic_cast<D*>(pb);
    D* pd2 = static_cast<D*>(pb);
}

```

如果pb真的指向一个D，那么pd1和pd2将取得同样的值，当pb==0时也是这样。但是，如果pb指向的（只）是一个B，那么`dynamic_cast`将能知道足够的信息去返回0，而`static_cast`则一定按照程序员对于pb确实指向D的判断，返回一个指向假定的D对象的指针。更糟的情况请考虑：

```

class D1 : public D { /* ... */ };
class D2 : public B { /* ... */ };
class X : public D1, public D2 { /* ... */ };

void g()
{
    D2* pd2 = new X;
    f(pd2);
}

```

这里的g()将用一个B（它并不是D的子对象）调用f()。因此，`dynamic_cast`将能正确地发现类型D的一个兄弟子对象。而`static_cast`则将返回一个指针，指向X的某个不适当的子对象。在我的记忆里，是Martin O'Riordan第一次使这种情况引起了我的注意。

14.3.3 `reinterpret_cast`运算符

记法`reinterpret_cast<T>(e)`是想取代`(T)e`，用在做例如`char*`到`int*`，或者`Some_class*`到`Unrelated_class*`的转换，这些东西从本质上说就是不安全的，与实现有关的。简单说，`reinterpret_cast`将返回一个值，对其参数做生硬粗鲁的重新解释。

例如：

```
class S;
class T;

void f(int* pi, char* pc, S* ps, T* pt, int i)
{
    S* ps2 = reinterpret_cast<S*>(pi);
    S* ps3 = reinterpret_cast<S*>(pt);
    char* pc2 = reinterpret_cast<char*>(pt);
    int* pi2 = reinterpret_cast<int*>(pc);
    int i2 = reinterpret_cast<int>(pc);
    int* pi3 = reinterpret_cast<int*>(i);
}
```

`reinterpret_cast` 运算符允许将任意指针转换到其他指针类型，也允许做任意整数类型和任意指针类型之间的转换。从本质上说所有这些转换都是不安全的，依赖于实现的，或两者都是。除非所希望的转换本身就是低级的和不安全的，否则程序员还是应该使用其他强制。

与 `static_cast` 不同，`reinterpret_cast` 的结果不能安全地用于其他目的，除非是转换回原来的类型。即使在最好的情况下，其他使用也是不可移植的。这就是为什么到函数的指针或到成员指针的转换都是 `reinterpret_cast`，而不是 `static_cast`。例如：

```
void thump(char* p) { *p = 'x'; }

typedef void (*PF)(const char*);
PF pf;

void g(const char* pc)
{
    thump(pc); // error: bad argument type
    pf = &thump; // error
    pf = static_cast<PF>(&thump); // error!
    pf = reinterpret_cast<PF>(&thump); // ok: on your
                                     // head be it
    pf(pc); // not guaranteed to work!
}
```

很清楚，让 `pf` 去指向 `thump` 是很危险的，因为这样做就是想欺骗类型系统，使它能允许将一个常量的地址传到某个要修改它的地方去。这也就是为什么我们必须使用强制的地方，特别是为什么必须使用“肮脏的”`reinterpret_cast` 的情况。许多人可能感到很意外，因为通过 `pf` 对 `thump` 的调用还是不能保证工作（在 C++ 里就像在 C 中一样）。这里的原因是，一个实现可以对不同函数类型使用不同的调用序列。特别是实现常常有很好的理由，对 `const` 和非 `const` 参数使用不同的调用序列。

请注意，`reinterpret_cast` 并不在类的层次中穿行。例如：

```
class A { /* ... */ };
class B { /* ... */ };
class D : public A, public B { /* ... */ };

void f(B* pb)
```

```

{
    D* pd1 = reinterpret_cast<D*>(pb);
    D* pd2 = static_cast<D*>(pb);
}

```

在典型情况下，这里pd1和pd2将得到不同的值。在调用：

```
f(new D);
```

pd2将指向传递来的D对象的开始位置，而pd1将指向该D对象的B子对象的开始。

`reinterpret_cast<T>(arg)`几乎与`(T) arg`同样糟糕。但是，无论如何，`reinterpret_cast`看得更清楚些，绝不在类层次中穿行，不会强制去掉`const`，因为其他强制提供了那些东西。`reinterpret_cast`是（仅有的）一个为执行低级操作而提供的运算符，通常用于做依赖于实现的转换。

14.3.4 `const_cast`运算符

在为老风格强制寻找替代品的过程中，最痛苦的就是要为`const`（常性）找到一种可以接受的处理方式。理想就是要保证这种“常性”绝不能不声不响地被删除掉。为了这个原因，`reinterpret_cast`、`dynamic_cast`和`static_cast`都被描述为保持常性的，也就是说，它们都不能用于“强制去掉`const`”。

记法`const_cast<T>(e)`就是想取代`(T)e`在这个方面的功能，用于通过转换获得对描述为`const`或`volatile`的数据的访问权。例如：

```

extern "C" char* strchr(char*, char);

inline const char* strchr(const char* p, char c)
{
    return strchr(const_cast<char*>(p), char c);
}

```

在`const_cast<T>(e)`里，类型`T`必须与参数`e`的类型一致，除了`const`或`volatile`修饰之外。转换的结果与`e`一样，只是类型变成`T`。

请注意，从一个原本就定义为`const`的对象出发，强制去掉`const`的结果是无定义的（13.3节）。

`const`（常性）保护的问题

不幸的是，在类型系统中存在一些微妙的东西，它们能在防止隐含地违背“常性”的保护上打开某种漏洞。考虑：

```

const char cc = 'a';
const char* pcc = &cc;
const char** ppcc = &pcc;
void* pv = ppcc; // no cast needed:
                  // ppcc isn't a const, it only points to one,
                  // but const vanished!
char** ppc = (char**)pv; // points to pcc

void f()
{
    **ppc = 'x'; // Zap!
}

```

不管怎样，允许`void*`不安全可以认为是能接受的，因为每个人都知道——至少是应该知道——从`void*`出发的强制本质上就是极难摆弄的事情。

如果你准备开始构造一些能保存各种各样指针类型的类（例如，为了使生成的代码最小化，参见15.5节），这种例子就会变得很有趣。

联合、通过使用省略号抑制对函数参数的类型检查，都是在防止隐含地违反“常性”的保护上面的漏洞。不管怎样，我更希望在一个系统里遗留的漏洞更少一些，不喜欢根本不提供保护的东西。像`void*`一样，程序员也应该知道，联合和不检查的函数参数从本质上就是危险的，只要可能就应该避免它们。如果真的需要，用的时候也必须特别小心。

14.3.5 新风格强制的影响

新风格的强制也是清除C++类型系统漏洞的长期努力的一部分，目标是尽可能减少不安全因素和容易出错的程序设计实践，并将它们局部化。在这里，我要讨论怎样处理与之相关的问题（老风格的强制、隐含的窄强制、以及函数风格的强制等），怎样将现存的代码转换为使用新强制运算符的代码。

1. 老风格强制

我的希望是，新风格强制能够完全取代(T)e记法。我已经建议贬低(T)e记法，希望委员会能给用户一个警告，说(T)e将很可能不再是C++标准某个未来版本的一个组成部分。我看到有两件事非常相像：这一件事情，还有就是将C++风格的函数原型引进ANSI/ISO C标准并打压不加检查的调用。但是上述想法并没有获得大多数的赞同，所以清理C++的事情可能永远也不会发生。

当然，更重要的是，新强制运算符为程序员个人和组织提供了一个机会，使他们在类型安全性比与C语言的向后兼容性更重要的情况下，能够避免老风格强制的不安全性。新风格强制还可以进一步由编译支持，方式就是对老风格强制提出警告。

新强制运算符开辟了一条指向程序设计的更安全风格的坦途，又没带来任何效率损失。这种东西的重要性应该能随着日月的流逝不断增长，随着代码普遍质量的改进和保证类型安全的工具的普遍使用——无论对C++，还是对其他语言。

2. 隐式窄转换

尽可能减少违反静态类型系统的情况，尽可能将这种违反的情况做得明显，这些想法都是新强制工作的基本点。很自然的，我也重新考虑了删除隐式窄转换的可能性，例如从`long`到`int`，从`double`到`char`（2.6.1节）。可惜，一般地禁止这些将是不可行的，只会产生相反的效果。主要问题在于算术运算可能溢出：

```
void f(char c, short s, int i)
{
    c++; // result might not fit in a char
    s++; // result might not fit in a short
    i++; // might overflow
}
```

如果我们禁止隐式窄转换，`c++`和`s++`都将变成非法的，因为在执行算术运算之前`char`和`short`都被提升到`int`。如果要求显式窄转换，这些东西都需要重写为：

```

void f(char c, short s, int i)
{
    c = static_cast<char>(c+1);
    s = static_cast<short>(s+1);
    i++;
}

```

我看，如果没有重大利益的话，强加上这种记法负担是没有任何希望的。但好处在哪里呢？在代码中到处散布显式的强制不会使代码更清晰，也不会减少错误的数量，因为人们将会不假思索地加上许多强制。`i++`同样也不安全，因为它也可能出现溢出。增加强制还可能产生相反的效果，因为本来实现有可能在运行时自动捕捉溢出情况，明显地使用强制就会抑制这种检查。另一种更好的方式应该是定义`dynamic_cast`，让它对数值运算对象都执行运行时的检查。按这种方式，那些认为检查很重要的用户就可以根据自己的经验，在认为实际中值得做这种检查的地方加`dynamic_cast`。换一种方式，人也可以直接写一个检查函数，并在需要时使用之，例如（15.6.2节）：

```

template<class V, class U> V narrow(U u)
{
    V v = u;
    if (v!=u) throw bad_narrowing;
    return v;
}

```

虽然禁止窄强制是不可行的（因为那将需要彻底翻查所有的指导算术运算的规则），但也还是存在一些应该反对的转换，实现中应该通过警告来要求进一步的确认：例如浮点类型到整数类型，`long`到`short`，`long`到`int`以及`long`到`char`。`Cfront`总要做这些事情。按照我的经验，其他潜在的窄转换，例如`int`到`float`和`int`到`char`则经常是无害的，反复做出警告将使用户无法接受。

3. 建构函数调用的记法

`C++`支持将建构函数记法`T(v)`作为老风格强制记法`(T)v`的同义词。一种更好的解决办法是将`T(v)`重新定义为合法对象构造的同义词，就像在下面这样的初始化里：

```
T val(v);
```

（对这种东西我们没有好的名字）。对这种改变也需要一个转变过程——就像有关贬低`(T)v`的建议一样——这将打破许多现存的代码。与贬低`(T)v`一样，这个建议也没有被委员会的大多数人接受。还好，那些希望使用显式形式（避免隐式转换）的人，比如说为消除歧义性，可以采用写类模板的方式做这件事情（15.6.2节）。

4. 新强制的使用

如果没有理解这里提出的各种微妙细节，也可能使用新强制吗？原来使用老风格强制的代码能转到使用新风格强制，而不会使程序员由于语言规则而陷入困境吗？为了使新强制比老强制更广泛地为人们所偏爱，对这两个问题的回答都必须是肯定的。

一种简单的转换策略就是对所有强制都用`static_cast`，然后再看编译程序会说些什么。如果对其中某些情况，编译程序不喜欢`static_cast`，那么这些情况就值得检查。如果问题是违反了`const`，那就去看强制的结果是否真正引起了违反的情况；如果不是，在这里就应

该用`const_cast`。如果问题是不完全的类型，指向函数的指针，或者不相干的指针之间的强制，首先应该试着弄清结果指针在使用之前确实已经强制回来了。如果问题是某些类似于指针与`int`间转换的情况，人们就应该努力地去想怎样才能继续下去。如果一个强制无法删除，那么，在这种情况下，`reinterpret_cast`将恰好能完成老风格强制所做的那种事情。

在大多数情况下，这个分析及其删除老风格强制的结果能用一个不太复杂的程序完成。当然，对于所有情况，最好是所有的强制——无论是老的新的——都能删除掉。

第15章 模板

没有任何东西比为事物建立新秩序更困难，
获得成功的希望更渺茫，
处理起来更危险。
——尼科罗·马基雅维利^Θ

支持参数化类型——类模板——对模板参数的限制——避免存储消耗——函数模板——函数模板参数的推断——函数模板参数的显式描述——模板中的条件——语法——组合技术——模板类之间的关系——成员模板——模板的实例化——模板里的名字约束——专门化——显式实例化——文件里的模板模型——模板的重要性。

15.1 引言

在关于“什么是”的文章 [Stroustrup, 1986b] (3.15节) 里就明确提出模板和异常是C++语言所需要的，这两个特征的设计出现在 [Stroustrup, 1988b] [Koenig, 1989b] [Koenig, 1990] 以及ARM里面，将它们放进语言中已经被明确地包含在有关C++标准化的建议书里。因此，虽然在C++里模板和异常处理的实现、能够使用都是在开始标准化工作之后的事，它们的设计以及对它们的渴求可以在C++历史中回溯到很久以前。

模板概念植根于对描述参数化容器类的愿望；异常来自于渴望为运行时的错误处理提供一种标准化的方式。对于这两种情况，从C语言继承来的机制在实际应用中都显得过于基本也过于简单。C的机制不能允许程序员直接表达自己的目的，也不能很好地与C++的各种关键特征互相协调使用。在带类的C最早的日子，我们曾经用宏机制实现过参数化的容器类 (2.9.2节)，但是，C语言的宏根本不遵守作用域规则和类型规则，也不能与工具友好相处。在早期C++里用于处理错误的机制，例如setjmp/longjmp以及错误指示 (如errno)，也不能很好地与建构函数和析构函数相互配合。

这些特征的缺乏导致了一些包装性的设计、不必要的低级编码风格，最终导致在综合使用来自不同提供商的库时出现问题。换句话说，没有这些特征，将使人们在维持一种一致性的 (高层) 抽象方面遇到不必要的麻烦。

在我的心里，模板和异常是一个硬币的两面：模板通过扩展静态类型检查所能处理的问题的范围，能够减少运行时错误出现的数量；异常就是为处理这些错误而提供的一种机制。模板使人有可能管理异常处理问题，方式是将运行时错误处理的需要减少到一些最基本的情况。异常使人们能够管理一般性的基于模板的库，因为它提供了一种方式，使这些库可以报告错误。

15.2 模板

在C++的原始设计中，就已经考虑了带参数类型的问题，但是这件事后来被推迟了，由于

^Θ Niccolo Machiavelli, 1469—1527, 意大利政治家兼历史学家。——译者注

没时间去做彻底的试验，去探究有关的设计和实现方面的问题；也由于害怕它们可能给实现增加新的复杂性。特别是我很担心某种不良设计可能导致极慢的编译和连接。我还认为，即使是一种能很好支持参数化类型的机制也会使移植系统的时间显著增加。不幸的是，我的这些担心都被证明是很有道理的。

模板被认为是在设计真正的容器时最关键的东西。我第一次是1988年在丹佛召开的USENIX C++会议发表了有关模板的设计[Stroustrup, 1988b]。我将问题总结为：

“在C++的上下文环境中，问题是：

- 1) 类型的参数化能不能很容易使用？
- 2) 使用参数化类型的对象能不能像“手工编码”的类的对象一样高效？
- 3) 能不能将参数化类型的一般形式集成到C++里？
- 4) 能不能将参数化类型实现好，使得其编译和连接的速度能够达到类似于不支持类型参数化的编译系统的水平？
- 5) 能不能使这种编译系统比较简单，容易移植？”

这些是我对于模板设计的检验准则。当然，我对所有这些问题的回答都是肯定的。我也用下面方式陈述了有关的基本设计选择：

“对许多人而言，使用C++最大的问题就是缺乏一个扩充的标准库。要产生这种库，遇到的最主要问题就是，C++没有提供一种充分一般的机制，以便于定义‘容器类’，如表、向量和关联数组等。存在两种提供这些类/类型的途径：

- 1) Smalltalk的途径：基于动态类型和继承。
- 2) Clu的途径：基于静态类型和一种为类型提供类型参数的功能。

前一种方式更灵活，但也会带来严重的运行时间代价，更重要的是，它将会使通过静态检查捕捉接口错误的所有企图完全落空。后一种方式在传统上也带来一些问题，使语言功能复杂了许多，也使编译/连接时的环境更慢、更费事。这种途径在灵活性方面也有局限性，因为在那些使用了它的语言里（值得提出的是Ada）根本就没有继承机制。

在理想上，我们最希望C++能够有一种机制，其结构类似于Clu的方式，具有理想的运行时间和空间需求，编译时的开销又很低。另一方面又能具有像Smalltalk那样的灵活性。前者是可能的，而对于许多重要情况，后者也是很接近的。”

这样，最关键的问题看起来就是记法上的方便性、运行时效率和类型安全性。最主要的限制是可移植性、合理的编译和连接效率——包括模板类和模板函数的实例化，无论是直接还是间接地使用在程序里面。

要确定我们到底需要什么样的参数化类型功能，最关键的事情就是用宏写出一些伪装的参数化类型。除了我之外，Andrew Koenig、Jonathan Shapiro和Alex Stepanov都写了许多模板风格的宏，以帮助确定为支持这种风格的程序设计到底需要什么样的语言机制。Ada的特征不太符合我对于模板机制的想法，它只是成为我厌恶模板实例化运算符的一个根源（15.10.1节）。Alex Stepanov对Ada有深入的了解，因此某些Ada风格有可能通过他的例子进入了我们的思想中。

最早将模板机制的实现集成到编译器中是在Cfront的一个版本里，它（只）支持类模板，是由Object Design公司的Sam Haradhvala在1989年写的。这个版本后来被Stan Lippman扩展为

一个完全的实现，由Glen McCluskey设计的模板实例化机制支持，这个机制从Tony Hansen、Andrew Koenig、Rob Murray和我 [McCluskey, 1992]得到了一些意见。Texas Instruments的Mary Fontana和Martin Neath写出了一个公开域的预处理器，实现了模板的一种变形 [Fontana, 1991]。

这些东西和其他的实现使我们取得了许多经验。但无论如何，我和许多其他人对于把一种并没有完全理解的东西放进标准里还是感到神经很紧张，因此，在ARM里定义的模板机制就有意弄得尽可能的小。在那时已经认识到这个东西可能是太小了，但是要删除一些不幸的特征远比增加一些需要的特征更困难得多。

在ARM里表述的模板机制在1990年七月被ANSI C++委员会接受。对于将模板机制接受到标准草稿中有一个重要的论据，那就是委员会成员在一个工作组里讨论有关问题时，看到在我们中间已经有了多于50万行采用模板和“伪造模板”的C++代码，而且它们正在实际使用之中。

回过头看，模板恰好成为精炼一种新语言特征的两种策略之间的分界线。在模板之前我一直是通过实现、使用、讨论、再实现的过程去精炼一个语言特征。而在模板之后，各种特征首先在委员会里进行广泛而深入的讨论，而实现通常是和这些讨论并行的。有关模板的讨论并没有像对它所应该做的那样广泛，我也缺乏批判性的实现经验。这就导致后来基于实现和使用经验又对模板进行了多方面的修订。在有关异常处理的扩充中，我又重新捡起了获得个人实现经验的实践，作为一种关键性的设计活动。

除了还有某些毛刺需要进一步修整外，模板确实能够完成我们预计它应该做的事情。特别是模板使人能够设计出高效的、紧凑的、类型安全的C++容器类，使用起来也很方便。如果没有模板，实现的选择就将不得不推到弱类型的或者是动态类型的方式上去，这些将对程序的结构和效率诸方面都造成伤害。

我确实认为，在开始描述模板机制时自己是过于谨慎和保守了。我们原来就应该把许多特征加进来，例如对函数模板参数的显式描述（15.6.2节）、有关非类型函数模板参数的推导（15.6.1节）以及嵌套模板（15.9.3节）等。这些特征并没有给实现者增加多少负担，但是却对用户特别有帮助。在另一方面，我也没有在模板实例化的领域（15.10节）里给实现者提供足够的指导和支持。我无法知道的是，如果我继续进行模板的设计，而没有参考那些实现和使用它们取得的经验，我是否就一定会做得更糟而不是更好。

这里的描述反映了事情的当前状态，反映了经过许多试验获得的经验，以及由ANSI/ISO标准化委员会通过的反映了这些经验的解决方案。这里的名字约束规则、显式的实例化机制、对于描述的限制、对模板函数的显式量化，都已经于1993年11月在San Jose通过，已经放进C++里，作为对模板定义的更普遍清理的一个组成部分。

有关模板的讨论按下面方式组织：

- 15.3节 类模板
- 15.4节 对模板参数的限制
- 15.5节 避免代码重复
- 15.6节 函数模板
- 15.7节 语法
- 15.8节 组合技术

- 15.9节 模板类之间的关系
- 15.10节 模板的实例化
- 15.11节 模板的作用

15.3 类模板

对这里的关键结构有如下解释 [Stroustrup, 1988b]:

“一个C++的参数化类型将被称为一个类模板。类模板可以描述如何构造出一些个别的类，其方式很像在类里描述如何构造起个别对象。一个向量的模板类可以像下面这样声明：

```
template<class T> class vector {
    T* v;
    int sz;
public:
    vector(int);
    T& operator[](int);
    T& elem(int i) { return v[i]; }
    // ...
};
```

前缀`template<class T>`说明了这里声明的是一个模板，一个类型`T`的参数类型将在声明中使用。在引入之后，在模板的作用域里，`T`就可以像其他类型名字一样使用了。向量模板可以像下面这样使用：

```
vector<int> v1(20);
vector<complex> v2(30);

typedef vector<complex> cvec; // make cvec a synonym
                             // for vector<complex>.
cvec v3(40); // v2 and v3 are of the same type.

void f()
{
    v1[3] = 7;
    v2[3] = v3.elem(4) = complex(7,8);
}
```

很清楚，类模板的使用并不比类更困难。类模板实例的完整名字读起来也很容易，例如`vector<int>`或者`vector<complex>`。人们甚至认为比它们语言内部的数组类型（例如`int[]`或者`complex[]`）更容易读。如果认为完整的名字太长，那就可以通过`typedef`引进缩写形式。

与类的声明相比，声明一个类模板并不复杂多少。关键字`class`用于指明类型参数的类型部分，部分地是因为它以很清楚的词的形式出现，部分地也是因为这样可以节约一个关键字。在这个上下文环境里，`class`的意思是‘任意类型’，而不仅是‘某种用户定义类型’。

在这里使用尖括号`<...>`而不是圆括号`(...)`，是为了强调模板参数具有不同的性质（它们将在编译时求值）。也是因为圆括号在C++里已经是过度使用了。

引进关键字`template`能使模板声明很容易看清楚（无论对人还是对工具）。同时也为模板类和模板函数提供了一种共有的语法形式。”

模板是为生成类型提供的一种机制。它们本身并不是类型，也没有运行时的表示形式，因此它们对于对象的布局没有任何影响。

要强调在运行时具有与宏一样的效率，一个原因是因为我希望模板在时间和空间上效率都足够高，也能用于数组或者表这样的低级类型。为了这些，我把在线机制作为最关键的东西。我特别把标准的数组和表模板看成能将C语言的低级数组概念禁闭起来的最现实的途径，这样才能将它放到实现的层次里去，在那里它能工作得很好。要想使高层次的另一些选择——比如说带有size()运算的检查范围的数组，多维数组，带有完善数值向量运算和复制语义的向量类型等——能够被用户接受，它们就必须在运行时间、空间和记法的方便性方面都能接近内部数组对应的各个方面。

换句话说，支持参数化类型的语言机制应该使谨慎的用户也能负担得起，愿意用标准库类（8.5节）代替数组的使用。自然，内部的数组还在那里，供需要它们的人之用，也是为了在千百万行上使用它们的老代码。无论如何，我的意图是，要为那些把方便和安全看得比兼容性更重要的人提供一种高效的替代品。

还有，C++支持虚函数，以它们作为一种变形，替代那种最明显的实现方式就是跳转表的概念。例如，一个T的“纯虚”集合可以实现为一个模板，它是一个带有在T对象上操作的虚函数的抽象类（13.2.2节）。由于这个原因，我感到应该将注意力集中到基于源代码正文和集中于编译时间的解决方案上，它们能提供接近最优的运行时间和空间性能。

非类型模板参数

除了类型参数之外，C++也允许非类型的模板参数。这种机制基本上被看作是为容器类提供大小和界限所需的信息。例如：

```
template<class T, int i> class Buffer {
    T v[i];
    int sz;
public:
    Buffer() :sz(i) {}
    // ...
};
```

在那些运行时效率和紧凑性非常要緊的地方，为了能与C语言的数组和结构竞争，这样的模板就非常要緊了。传递大小信息将允许实现者不使用自由空间。

如果不能使用非类型的参数，用户就必须把关于大小的信息编码用到数组类型里。例如：

```
template<class T, class A> class Buf {
    A v;
    int sz;
public:
    Buf() :sz(sizeof(A)/sizeof(T)) {}
    // ...
};
```

```
Buf<int,int[700]> b;
```

这样看起来很不直接，也容易出错，而且很明显的无法拓展到整数之外的类型。特别是我还希望有指向函数模板参数的指针，能提供进一步的灵活性。

在模板的初始设计中，不允许用名字空间或模板作为模板的参数。这些限制是过于谨慎

的又一个案例。我现在看不出有任何理由去禁止这种参数，它们无疑是很有用的。以类模板作为模板参数在1994年3月的圣迭戈会议上获得通过。

15.4 对模板参数的限制

对模板参数并没有提出任何限制。相反，所有类型检查都被推迟到模板实例化的时刻进行 [Stroustrup, 1988b]：

“‘应该要求用户去描述一集操作，用以说明什么样的类型能够用作某个模板类型的参数吗？’ 例如：

```
// The operations =, ==, <, and <=
// must be defined for an argument type T

template <
    class T {
        T& operator=(const T&);
        int operator==(const T&, const T&);
        int operator<=(const T&, const T&);
        int operator<(const T&, const T&);
    };
>
class vector {
    // ...
};
```

不！要求用户提供这样的信息就会降低参数机制的灵活性，又不会使实现变得更简单，或使这种功能更安全……。曾经有意见说，如果对类型参数给出了完整的运算集合，阅读和理解参数化类型将会更容易些。我看这里面有两个问题：这种列表通常会变得很长，以至变得很不容易阅读和理解；此外，在许多应用中都将需要大量的模板。”

回头再看，我理解了这些限制对于可读性和早期错误检测的重要性。但是我也发现了表述限制的另外一些问题：函数类型对于有效的限制而言显得太特殊了。如果从字面上看，函数类型将对解决的方案生产过分限制。以vector为例，人们或许认为任何能接受两个T类型参数的 < 都可以接受。但是，除了内部的 < 运算符之外，我们可能还有许多其他候选者：

```
int X::operator<(X);
int Y::operator<(const Y&);
int operator<(Z,Z);
int operator<(const ZZ&,const ZZ&);
```

如果从字面上看，好像只有ZZ能成为vector可以接受的参数。

限制模板的想法总是重复地出现：

— 有些人认为如果模板参数是受限制的，那么就可能生成更好的代码——我不相信这一点。

— 有些人认为如果缺乏限制，静态类型检查将会受到影响——不会，但是某些静态类型检查将被推迟到连接的时候，而这确实是一个实际问题。

— 有些人认为如果有了限制，模板声明将更容易理解——经常，确实有这种情况。

在下面两小节里要给出描述限制的另外两种方式。只有在实际需要时才生成成员函数（15.5

节) 和特殊化 (15.10.3节) 是另外的能在某些情况下提供限制的方法。

15.4.1 通过派生加以限制

Doug Lea、Andrew Koenig、Philippe Gautron、我和其他许多人都独立地发现了采用继承的语法描述限制的技巧。例如：

```
template <class T> class Comparable {
    T& operator=(const T&);
    int operator==(const T&, const T&);
    int operator<=(const T&, const T&);
    int operator<(const T&, const T&);
};

template <class T : Comparable>
class vector {
    // ...
};
```

这确实很有意义。各种各样的建议在细节上有所差别，但它们都有思想中的作用，能够将检查错误和报告的时间提前到对独立编译单元进行编译的时候。沿着这个思路走下去的建议还在C++标准化组里讨论。

但是，我则从根本上反对通过派生去描述限制。因为它将鼓励程序员按这种方式去组织程序，任何能成为限制的东西都可能变成一个类，这就会鼓励通过继承去描述所有的限制。例如，不是去说“T必须有一个小于函数”，而是必须说“T必须是从Comparable派生的”。这是一种间接的相当灵活的描述限制的方式，但也很容易导致对继承的过度使用。

因为在内部类型 (例如int和double) 之间不存在继承关系，派生方式就不能用于对这些类型的限制。类似地，用派生将无法描述同时适用于用户定义类型和内部类型的限制。例如，int和complex常常同时是可以接受的模板参数，这个情况就无法通过派生的方式描述。

进一步说，写模板的人不可能预见到模板的全部应用。这就导致程序员在初始时给模板参数加上了过分的限制，而到后来——由于经验——又放松了对它们的限制。通过派生加以限制的逻辑推论是引进一个最一般的基类，描述“没有任何限制”。而无论如何，这样的一个基类将成为许多遍历代码的发源地，无论是在模板的上下文里面，还是在其他的地方 (参见14.2.3节)。

通过派生去限制模板参数，也不可能处理另一个已被发现是非常讨厌的问题：派生限制将不允许程序员写出两个具有同样名字的模板，以便用一个处理指针类型，另一个处理非指针类型。Keith Gorlen的提醒第一次使我注意到了这个问题。它可以通过模板函数重载的方式去处理 (15.6.3节)。

对这种途径的另一个更具根本性的批评是，它将继承机制用到了某些本质上并不是子类型的事情上。我把用继承表述限制看成是一个例子，在这里，使用继承机制只是因为它是一种时尚，而不是有什么更根本性的理由。继承关系并不仅仅是一种有用的关系。换句话说，也不是类型之间的任何关系或者有关类型的所有陈述都应该削足适履，硬塞进继承的框架之中。

15.4.2 通过使用加以限制

当我第一次接触模板的实现问题时，就使用了在在线函数里描述限制的方式去解决限制

问题。例如：

```
template<class T> class X {
    // ...
    void constraints(T* tp)
    {
        // T must have:
        B* bp = tp; // an accessible base B
        tp->f(); // a member function f
        T a(1); // a constructor from int
        a = *tp; // assignment
        // ...
    }
};
```

可惜的是，这种方式借助了某些局部的实现细节：Cfront在实例化模板声明的时候，对所有在线函数作完全的语法和语义检查。但实际上，如果不调用，那么根本就不需要对一集特定的模板参数生成相应的函数版本（15.5节）。

这种方式使写模板的人能去描述一个限制函数，而模板的用户如果觉得这种检查很方便，就可以通过调用这个函数去检查有关的限制。

如果写模板的人不想打扰用户，他也可以从每个建构函数里调用constraints()。当然，如果建构函数很多，而且这些函数又不能正常地做在线处理，这种做法就可能变得非常讨厌了。

如果需要的话，这个概念也可以形式化为一个语言特征：

```
template<class T>
constraints {
    T* tp; // T must have:
    B* bp = tp; // an accessible base B
    tp->f(); // a member function f
    T a(1); // a constructor from int
    a = *tp; // assignment
    // ...
}
class X {
    // ...
};
```

这将允许对模板参数的函数模板进行限制。不过我还是很怀疑这种扩充的价值。这也是我见到过的惟一比较接近我的想法的限制系统，它能在保留了充分一般、比较紧凑、容易理解和容易实现的同时，又不会给模板参数加上过分的限制。

参见15.9.1节和15.9.2节，那里有通过使用模板函数表示基本限制的一些例子。

15.5 避免代码重复

避免由于做的实例化太多而造成不必要的空间开销，这件事一直被当作第一个层次上的问题——也就是说设计和语言层的问题——而不是实现的细节问题。规则要求模板函数（15.10节，15.10.4节）的“迟”实例化，以保证在不同编译单元里使用同样模板参数时不会出现重复的代码。我认为它不大靠得住，那些早先（甚至很晚）的模板实现都应该能去查看一个类对不同模板参数的实例化，并确定哪些实例代码的全部或者部分是可以共享的。现在我还认为这是最重要的东西，能够避免由于代码重复而出现的肿胀——就像宏展开，或者像

其他采用了实例化机制原语的语言中所遇到的那样 [Stroustrup, 1988b]:

“除了其他的东西，派生（继承）还保证了不同类型间的代码共享（非虚基类的函数代码由它的派生类所共享）。一个模板的不同实例不能共享代码，除非采用了某种更聪明的编译策略。我还看不到最近会有能投入使用的这类聪明东西。那么，派生能用于削减由于使用模板而产生的代码重复问题吗？这就涉及到由常规的类派生出模板的问题。例如 [Stroustrup, 1988b]:

```
template<class T> class vector { // general vector type
    T* v;
    int sz;
public:
    vector(int);
    T& elem(int i) { return v[i]; }
    T& operator[](int i);
    // ...
};

template<class T> class pvector : vector<void*> {
    // build all vector of pointers
    // based on vector<void*>
public:
    pvector(int i) : vector<void*>(i) {}
    T*& elem(int i)
        { return (T*)& vector<void*>::elem(i); }
    T*& operator[](int i)
        { return (T*)& vector<void*>::operator[](i); }
    // ...
};

pvector<int> pivec(100);
pvector<complex> icmpvec(200);
pvector<char> pcvec(300);
```

这三个指针的向量类的实现将完全是共享的。它们的实现完全是通过派生和基于类 `vector<void*>` 的实现的在线展开。`vector<void*>` 的实现将是标准库的一个极好候选。”

已经证明，这种技术能够在实际使用中约束代码的肿胀。没有使用这类技术的人（无论是在C++里；还是在其他有类似的类型参数化功能的语言里）都发现，即使是对中等大小的程序，重复代码也能造成成兆字节的代码空间代价。

面对这种忧虑，我认为能够让实现只去实例化那些实际使用的模板函数是非常重要的事情。例如，给定一个带有函数 `f` 和 `g` 的模板 `T`，如果对于某个给定的模板参数 `g` 根本不会使用，实现就应该能做到只完成 `f` 的实例化。

我也觉得，只按照给定的模板参数对被调用的函数生成模板函数的实际版本，这实际上又增加了另一个级别的重要的灵活性 [Stroustrup, 1988b]:

“考虑 `vector<T>`，为了提供一个排序操作，我们必须要求类型 `T` 具有某种顺序关系。并不是所有的类型都有这种东西。如果在 `vector<T>` 的声明中必须描述 `T` 的运算集合，那么就必须定义两个向量类型：一个针对那些带有序关系的类型的对象；另一个则针对那些没有序关系的类型。如果在 `vector<T>` 的声明中不必描述 `T` 的运算集合，那么就可以只有一个向量

类型。很自然，如果某个类型glob没有一个有序关系的话，人们就没有办法对这个类型的对象的向量排序。如果要试图这样做，编译程序应该拒绝生成对应的排序函数vector<glob>::sort()。”

15.6 函数模板

除了类模板之外，C++还提供了函数模板。引进函数模板，部分地是因为我们已经很清楚，需要有模板类的成员函数，还因为如果没有这种东西，模板的概念看起来就不够完全。自然，在这里我们也要引用几个教科书性质的例子，例如sort()函数等。Adrew Koenig和Alex Stepanov是这些要求函数模板的例子的主要贡献者。数组排序被看作是最基本的例子：

```
// declaration of a template function:
template<class T> void sort(vector<T>&);

void f(vector<int>& vi, vector<String>& vs)
{
    sort(vi);    // sort(vector<int>& v);
    sort(vs);    // sort(vector<String>& v);
}

// definition of a template function:
template<class T> void sort(vector<T>& v)
/*
    Sort the elements into increasing order

    Algorithm: bubble sort (inefficient and obvious)
*/
{
    unsigned int n = v.size();

    for (int i=0; i<n-1; i++)
        for (int j=n-1; i<j; j--)
            if (v[j] < v[j-1]) { // swap v[j] and v[j-1]
                T temp = v[j];
                v[j] = v[j-1];
                v[j-1] = temp;
            }
}
}
```

如我们的预期，函数模板本身也已经被证明是很有用的。它们也被证明是支持类模板的重要机制，对某些服务而言，采用非成员函数比用成员函数更合适（例如友元，见3.6.1节）。

下面几小节将考察与模板函数有关的技术细节。

15.6.1 函数模板参数的推断

对函数模板，我们并不需要明显描述模板参数。如上所示，编译程序能够从调用的实际参数将它们推导出来。自然，每个没有显式描述的模板参数（15.6.2节）都必须能由某个函数参数惟一地确定，这就是原来手册里所说的全部东西。在标准化过程中事情变得很清楚：在从实际函数参数推断出模板参数类型方面，必须把编译程序应该多么能干这件事情严格地描述好。例如，下面这个程序合法吗？

```

template<class T, int i>
    T lookup(Buffer<T,i>& b, const char* p);

int f(Buffer<int,128>& buf, const char* p)
{
    return lookup(buf,p); // use the lookup() where
                           // T is int and i is 128
}

```

过去的回答一直是否定的，因为无法推断出非类型参数。这也是一个实际问题，因为这意味着你不能定义非在线的非成员函数，让它在一个模板类上操作，而这个类又有一个非类型的模板参数。例如：

```

template<class T, int i> class Buffer {
    friend T lookup(Buffer&, const char* );
    // ...
};

```

这要求一个原先明显非法的模板函数定义。

在一个模板函数参数表里，可接受结构的检查表是：

```

T
const T
volatile T
T*
T&
T[n]
some_type[I]
CT<T>
CT<I>
T (*) (args)
some_type (*) (args_containing_T)
some_type (*) (args_containing_I)
T C::*
C T::*

```

这里的T是模板的一个类型参数，I是模板的一个非类型参数，C是一个类名字，CT则是一个前面已声明的类模板，args_containing_T是一个参数表，根据它，应用这些规则就能够确定T。这就使lookup()例子合法了。幸运的是，用户并不需要记住这个表，因为它不过是形式地写出明显的推导规则。

再看一个例子：

```

template<class T, class U> void f(const T*, U(*)(U));

int g(int);

void h(const char* p)
{
    f(p,g); // T is char, U is int
    f(p,h); // error: can't deduce U
}

```

请注意f()第一个调用的参数，我们很容易推导出有关的模板参数。再看f()的第二个调用，可以看到h()与模式U(*)(U)不匹配，因为h()的参数和返回值类型不同。

在弄清这个问题和其他类似的问题方面, John Spicer提供了很多帮助。

15.6.2 描述函数模板的参数

在刚开始做模板设计时, 我觉得应该允许明确地为模板函数描述模板参数, 可以采用与为模板类提供模板参数的同样方式。例如:

```
vector<int> v(10); // class, template argument 'int'
sort<int>(v);      // function, template argument 'int'
```

但是我后来拒绝了这个想法, 因为对于大部分例子, 根本不需要去显式地描述模板参数。我也害怕“含糊”和语法分析中的问题。例如, 我们该如何对下面例子进行语法分析?

```
void g()
{
    f<1>(0); // (f) < (1>(0)) or (f<1>) (0) ?
}
```

我现在不再认为这是个问题了。如果f是个模板名, f< 就是一个量化模板名的开始, 随后的东西就必须根据这一点进行解释; 如果f不是, 那么 < 就是小于号。

说显式描述有用, 是因为我们可能无法从一个模板函数的调用出发去推导出返回类型:

```
template<class T, class U> T convert(U u) { return u; }
void g(int i)
{
    convert(i);           // error: can't deduce T.
    convert<double>(i);  // T is double, U is int.
    convert<char,double>(i); // T is char, U is double.
    convert<char*,double>(i); // T is char*, U is double
                           // error: cannot convert
                           // a double to a char*.
}
```

就像函数的默认参数一样, 在一个显式的模板参数表里, 只有最后的参数是可缺省的。

能显式描述模板参数, 将使我们可以定义一些转换函数和对象创建函数。那些能做隐式转换函数能做的事的显式转换函数, 例如convert(), 也经常是很需要的东西, 这种转换函数也适合作为库的候选。这类东西的另一种变形是使用一个检查, 保证窄转换将产生运行时错误(14.3.5节)。

新强制运算符(14.3节)的语法形式和显式量化模板函数调用的语法相同, 这也是有意做出的选择。新强制表示的是一种不能用其他语言特征表述的操作。与它们类似的操作, 例如convert(), 可以表示为函数模板, 因此不必作为内部运算符。

显式描述函数模板参数还有另一个应用, 那就是通过描述有关类型或者局部变量的值, 去控制某些需要使用的算法。例如:

```
template<class TT, class AT> void f(AT a)
{
    TT temp = a; // use TT to control
                  // precision of computation
    // ...
}
```

```

void g(Array<float>& a)
{
    f<float>(a);
    f<double>(a);
    f<Quad>(a);
}

```

函数模板参数的显式描述于1993年11月在San Jose会议上被通过加进C++。

15.6.3 函数模板的重载

有了函数模板之后，就必须重新解决重载的处理问题。为避免引起语言定义的麻烦，我的选择是对模板参数只允许准确的匹配，在解析重载问题时更偏向于有同样名字的常规函数：

“具有同样名字的模板函数和其他函数的重载解析分三个步骤进行 [ARM]：

1) 在函数中查找准确的匹配 [ARM, 13.2节]；如果找到就调用它。

2) 查找这样的函数模板，从它出发能够生成出可以通过准确匹配进行调用的函数；如果找到就调用它。

3) 试着去做函数重载的解析 [ARM, 13.2节]，如果能找到这样的函数就调用它。

如果无法找到一个能够调用的函数，那么这个调用就是错误。在上述各种情况中，如果在某一步发现了多于一个能匹配的候选者，这个调用就是歧义的，因此也是错误的。”

回头看，这个规定大概是限制得太过分，也太专用。虽然它可以用，但却也为许多小的使人诧异和烦恼的问题打开了大门。

即使在那时事情也已经很清楚，最好是能采用某种方式，把对常规函数和模板函数的规则统一起来，只是我不知道怎么做。下面是Andrew Koenig提出的一种替代方式的梗概：

“对于一个调用，首先找到可能被调用的所有函数的集合，一般地说这里也可能包括从不同模板生成的函数。而后将普通的重载解析规则应用到这集函数上。”

这样就能够允许对模板函数参数的类型转换，也为所有的函数重载提供了一种共同框架。例如：

```

template<class T> class B { /* ... */ };
template<class T> class D : public B<T> { /* ... */ };

template<class T> void f(B<T>*);

void g(B<int>* pb, D<int>* pd)
{
    f(pb); // f<int>(pb)
    f(pd); // f<int>((B<int>*)pd); standard conversion used
}

```

必须让模板函数与继承之间能够正确地相互作用。还有：

```

template<class T> T max(T, T);

const int s = 7;

void k()

```

```

    max(s,7); // max(int(s),7); trivial conversion used
}

```

在ARM里实际上已经预料到需要放松有关的规则，允许模板函数的参数转换。现在的许多实现都允许上面这种例子。但无论如何，这个问题还留在那里，等待着正式的解决方案。

模板里的条件

在写模板函数时常常会感到有一种诱惑，想让写出的定义依赖于模板参数的性质。例如下面这段话 [Stroustrup, 1988b]：

“让我们考虑为向量类型提供一个打印函数，如果向量能够排序的话，就首先将它排好序，而后再打印。可以考虑提供一种机制，它能够询问对于给定类型的对象是否能够执行某种操作（比如说<操作）。例如：

```

template<class T> void vector<T>::print()
{
    // if T has a < operation sort before printing

    if (?T::operator<) sort();
    for (int i=0; i<sz; i++) { /* ... */ }
}

```

如果待打印向量的元素可以比较，那么就在打印向量的元素之前调用sort()，对不能比较的对象就不这么做。”

我决定反对提供这种类型询问功能，因为我那时——现在依然——明确认为，这将导致结构上极坏的代码。这种技术是宏爱好者们最坏的方面，在加上对类型获取的过分信赖（14.2.3节）。

与此相反的方法是通过对各种特殊的模板参数类型分别提供几个版本（15.10.3节），以实现专门化。另一种方式是把那些不能保证每个模板参数类型都有的操作孤立出来，放到另外的成员函数里，只在有关类型中确实有这些操作时才调用它们（15.5节）。最后，还可以采用重载函数模板的方式，为不同类型提供不同的实现。作为一个例子，让我给出一个reverse()函数模板，如果给它指明需要考虑的第一个和最后一个元素的遍历器，它就能翻转容器里元素的顺序。用户代码将具有下面的样子：

```

void f(ListIter<int> l1, ListIter<int> l2, int* p1, int* p2)
{
    reverse(p1,p2);
    reverse(l1,l2);
}

```

在这里，将用ListIter访问某种用户定义容器的元素，用int*访问常规整数数组的元素。为了做到这些，必须设法为两个调用选择reverse()的不同实现。

函数模板reverse()简单地基于其参数的类型去选择实现：

```

template <class Iter>
inline void reverse(Iter first, Iter last)
{
    rev(first,last,IterType(first));
}

```

通过重载解析选择IterType:

```
class RandomAccess { };

template <class T> inline RandomAccess IterType(T*)
{
    return RandomAccess();
}

class Forward { };

template <class T> inline Forward IterType(ListIter<T>)
{
    return Forward();
}
```

在这里, `int*`将选择`RandomAccess`, 而`ListIter`将选择`Forward`。通过这些遍历器的类型, 反过来又能确定应该使用的`rev()`版本:

```
template <class Iter>
inline void rev(Iter first, Iter last, Forward)
{
    // ...
}

template <class Iter>
inline void rev(Iter first, Iter last, RandomAccess)
{
    // ...
}
```

请注意, `rev()`的第三个参数实际上根本不用, 它的作用就是为重载机制提供帮助。

这里的基本观点是: 类型或算法的任何性质都可以用一个类型表示 (或许需要为此目的而专门定义)。在做好这些之后, 这样的类型就可以用于指导重载的解析, 通过它们把依赖于有关性质的函数选出来。除非这种服务于选择的类型本身也表示了一个基本性质, 否则这种技术就不够直接, 但一般说它是很有效的。

请注意, 借助于在线机制, 这种解析完全能在编译过程中完成, 因此将能直接调用正确的`rev()`函数, 不存在任何运行时的开销。还请注意, 这种机制还具有可扩充性, 当需要增加新的`rev()`实现的时候, 根本不必触动老的代码。这个例子是基于从Alex Stepanov [Stepanov, 1993] 那里来的一些思想。

如果所有其他方式都不行, 有时还可以借助于运行时的类型识别机制 (14.2.5节)。

15.7 语法

和以往一样, 语法总是一个问题。开始时我希望采用一种语法形式, 把模板参数直接放在模板名字的后面:

```
class vector<class T> {
    // ...
};
```

但是这种方式无法很清晰地扩展到函数模板 [Stroustrup, 1988b]:

“初看起来，不另外使用关键字的函数语法似乎更好一些：

```
T& index<class T>(vector<T>& v, int i) { /* ... */ }
```

但它们的使用不能与类模板并行不悖，因为函数模板的参数通常并不需要显式地描述：

```
int i = index(vi, 10);
char* p = index(vpc, 29);
```

无论如何，这种“比较简单”的语法也存在很烦人的问题：它过于精巧，在程序里很难看出这种形式的模板声明，因为模板参数深深地嵌在函数和类声明的语法中，对某些函数模板做语法分析有可能成为小小的噩梦。当然，可以写出这样的C++分析程序，使之能够处理上面index()这样的函数模板声明，其中模板参数的使用出现在定义之前。我知道这一点，因为我已经写了一个这种东西。但这件事不好做，看起来也不是很容易在传统编译技术中解决的问题。回头看，我觉得要是不使用关键字，不要求模板参数总在使用之前声明的话，结果会造成一系列问题，实际上与那些过于精巧、互相缠绕的C和C++声明语法造成的问题很类似。”

用最后的模板语法，index()的声明变成：

```
template<class T> T& index(vector<T>& v, int i) { /* ... */ }
```

在那时我也很严肃地讨论过，认为可以提供一种语法，使函数的返回值类型可以放在参数表之后，例如：

```
index<class T>(vector<T>& v, int i) return T& { /* ... */ }
```

或

```
index<class T>(vector<T>& v, int i) : T& { /* ... */ }
```

这将很好地解决语法分析问题。但是大部分人宁愿要一个关键字来帮助识别模板，这就使向这个方向的考虑都成为多余的了。

选择尖括号<...>而不用圆括号，是因为用户发现这样更容易读，也因为圆括号在C和C++里已经使用过度。Tom Pennello证明，在这个地方用圆括号，做语法分析也不困难。但这并没有改变人们的观点，因为（作为人的）读者[⊖]喜欢<...>。

还有一个问题也很讨厌：

```
List<List<int>> a;
```

这看起来是声明了一个整数表的表，而实际上它却是个语法错误，因为>>（右移或者输出）与两个单独的>>是不一样的。当然，通过一个简单的词法技巧就可以解决这个问题，但我还是希望保持语法和词法分析的清晰性。现在我看到这种错误出现得太频繁，听到了太多有关类似这种东西的抱怨：

```
List<List<int>> a;
```

这就使我很希望能通过某些努力，使这个问题彻底消失。我觉得听到用户的抱怨远比听到语言专家的抱怨更使人感到痛苦。

[⊖] reader，在英文中不能说一定是指人（读者），同样可以用于指处理源文件的分析程序等。作者在这里强调的是因为人们不喜欢，而不是程序做不出来。这是一种诙谐的说法。——译者注

15.8 组合技术

模板能支持一些安全而又强有力地组合技术。例如，模板可以递归地使用：

```
List<int> li;
List< List<int> > lli;
List< List< List<int> > > llli;
```

如果需要特定的“组合类型”，可以通过派生，以特殊化的方式将它们定义出来：

```
template<class T> class List2 : public List< List<T> > { };
template<class T> class List3 : public List2< List<T> > { };

List2<int> lli2;
List3<int> llli3;
```

这是派生的一种不太寻常的应用，因为在这里并不增加成员。这种派生的应用不会造成任何时间或者空间开销，它仅仅是一种组合技术。如果不能在组合中使用派生技术，那么就必须用某些特定的组合技术去扩大模板，否则这个语言就太可怜了。派生和模板之间平滑的交互作用一直是我感到惊喜的一个源泉。

这种组合类型的变量可以像对应的具有显式定义的类型一样使用，但反过来却不行：

```
void f()
{
    lli = lli2; // ok
    lli2 = lli; // error
}
```

这是因为公用派生定义的是一种子类型关系。

要想允许在两个方向上赋值，就需要对语言做一种扩充，引进真正的参数化同义词。例如：

```
template<class T> typedef List< List<T> > List4;

void (List< List<T> >& lst1, List4& lst2)
{
    lst1 = lst2;
    lst2 = lst1;
}
```

这种扩充在技术上是非常简单，但是我却无法确定，再引进一种只不过是重命名的新机制是否为一个明智之举。

在定义新类型时，派生机制也允许仅仅部分地给出模板参数的描述：

```
template<class U, class V> class X { /* ... */ };
template<class U> class XX : public X<U,int> { };
```

一般说，从模板类出发的派生提供了一种可能性，使人可以依据适应于派生类的信息去剪裁基类。这提供了一些特别有力的组合模式。例如：

```
template<class T> class Base { /* ... */ };
class Derived : public Base<Derived> { /* ... */ };
```

通过这种技术，我们能将派生类的信息嵌入基类之中。又见14.2.7节。

15.8.1 表达实现的策略

派生和模板在组合方面的另一项应用是作为一种技术，传递由对象所表示的实现策略。例如，有关排序中比较的意义、对于一个容器类的分配/释放存储的意义等，都可以通过模板参数提供 [2nd]：

“一种方式是使用模板，从所需要容器的界面和一个采用了 [2nd, 6.7.2节] 中描述的放置技术的分配器类组合出一个新的类：

```
template<class T, class A> class Controlled_container
: public Container<T>, private A {
    // ...
    void some_function()
    {
        // ...
        T* p = new(A::operator new(sizeof(T))) T;
        // ...
    }
    // ...
};
```

在这里使用一个模板是必需的，因为我们是在设计一个容器。需要从Container出发做派生，以便使Controlled_container能够作为容器使用。模板参数A的使用也是必需的，这样才能允许使用各种各样的分配器。例如：

```
class Shared : public Arena { /* ... */ };
class Fast_allocator { /* ... */ };
class Persistent : public Arena { /* ... */ };

Controlled_container<Process_descriptor, Shared> ptbl;
Controlled_container<Node, Fast_allocator> tree;
Controlled_container<Personnel_record, Persistent> payroll;
```

这是为派生类提供比较复杂的信息的一种通用策略。它的优点是非常系统化，而且允许以在线方式使用。当然，这样做也倾向于产生极长的名字。当然，还是可以用typedef为过长的类型名引进同义词。”

在Booch组件 [Booch, 1993] 中广泛地使用了这些组合技术。

15.8.2 描述顺序关系

现在考虑一个排序问题：我们有一个容器模板，有一个元素类型，有一个能够基于元素值对容器排序的函数。

我们不能把排序用的比较准则直接编码到容器类里，因为一般说，容器不应该把自己的需要强加到元素类型上。我们也不能将排序用的比较准则直接编码到元素类型里，因为被处理的元素可能存在多种排序方式。

这样，排序准则就既不在容器里编码，也不在元素类型里编码。相反，这个准则应该在要做特定操作的时候提供。例如，给定了一些表示瑞典人名字的字符串，什么是我应该用于

整理的比较准则呢？常用的对瑞典人名排序的整理方式有两种。自然，无论是一个通用的字符串类型，还是一个通用的排序算法，要求在它们定义时就必须知道对瑞典人名字排序的规则是不合适的。

因此，任何一个涉及到排序算法的一般性解决方案都要以某种通用的方式描述，使它的定义不但能处理任意一个特定类型，还能处理特定类型的特定使用。举个例子，让我们看看如何将标准库函数strcmp()推广到能用于任何类T的串。

首先我定义一个模板类，为类T的两个对象的比较确定一种默认的意义：

```
template<class T> class CMP {
public:
    static int eq(T a, T b) { return a==b; }
    static int lt(T a, T b) { return a<b; }
};
```

模板函数compare()比较basic_string，以上述形式的比较作为默认方式：

```
template<class T> class basic_string {
    // ...
};

template<class T, class C = CMP<T> >
int compare(const basic_string<T>& str1,
            const basic_string<T>& str2)
{
    for(int i=0; i<str1.length() && i< str2.length(); i++)
        if (!C::eq(str1[i],str2[i]))
            return C::lt(str1[i],str2[i]);
    return str2.length()-str1.length();
}

typedef basic_string<char> string;
```

有了成员模板（15.9.3节）之后，我们还可以换个方式，将compare()定义为basic_string的成员。

如果某人希望一个C<T>忽略大小写，或者希望它能反映某种本地化的情况，或者想返回两个用!C<T>::eq()比较的元素中较大一个的unicode值，等等，这些都可以通过基于T的“本地”操作来完成，只要适当地给出对C<T>::eq()和C<T>::lt() 的定义。这就使我们可以通过由CMP提供的操作表达任何（比较、排序等）算法，也就能够表示我们所需要的容器了。例如：

```
class LITERATE {
    static int eq(char a, char b) { return a==b; }
    static int lt(char, char); // use literary convention
};

void f(string swedel, string swede2)
{
    compare(swedel, swede2); // ordinary/telephone order
    compare<char,LITERATE>(swedel, swede2); // literary order
}
```

我用模板参数的方式传递比较准则，因为这是一种不增加任何运行时代价而传递一些操作的

方式。特别是比较操作`eq()`和`lt()`很容易做成在线。我还用了一个默认参数，以避免给所有的人增添记法上的代价。这种技术还有一些变形，可以在[2nd, 8.4节]找到。

另一个不那么深奥的例子（对非瑞典人来说）是做关心或不关心大小写的比较：

```
void f(string s1, string s2)
{
    compare(s1,s2); // case sensitive
    compare<char,NOCASE>(s1,s2); // not sensitive to case
}
```

请注意，这里的CMP模板类绝不会用于定义对象，它的所有成员都是`static`和`public`，因此它应该是个名字空间（第17章）：

```
template<class T> namespace CMP {
    int eq(T a, T b) { return a==b; }
    int lt(T a, T b) { return a<b; }
}
```

可惜的是，名字空间模板（还）不是C++的组成部分。

15.9 模板类之间的关系

把模板理解为一个规范，认为它说明了一些特定的类将如何创建起来，这种观点也是很有价值的。换句话说，模板实现就是一种基于用户所描述的需要去生成类型的机制。

就C++语言规则而论，由同一个类模板生成的两个类之间没有任何关系。例如：

```
template<class T> class Set { /* ... */ };

class Shape { /* ... */ };
class Circle : public Shape { /* ... */ };
```

有了这些声明之后，人们有时会想把`Set<Circle>`当作`Set<Shape>`，或者把`Set<Circle*>`当作`Set<Shape*>`。例如：

```
void f(Set<Shape>&);

void g(Set<Circle>& s)
{
    f(s);
}
```

这不能通过编译，因为不存在内部的从`Set<Circle>&`到`Set<Shape>&`的转换。实际上也确实不该有；将`Set<Circle>`当作`Set<Shape>`的考虑是一个根本性的——但并非不常见——的概念错误。特别是`Set<Circle>`能够保证自己的成员是`Circle`，因此用户可以对它的成员安全而高效地使用`Circle`的特殊操作，例如去确定它们的半径等等。如果我们允许将一个`Set<Circle>`当作一个`Set<Shape>`看待，我们就不再继续有这些保证了，因为将任何`Shape`，例如`triangle`，放进`Set<Shape>`都已假定是可以接受的。

15.9.1 继承关系

这也说明，由同一个模板生成的类之间并不存在任何默认的关系。当然，有时我们可能希望存在某种关系。我考虑过是否需要有一个特殊操作来表述这类关系，但还是决定不要它，

因为许多有用的转换都可以被表示为继承关系，或者可以通过常规的转换运算符表述。但是，无论如何，也确实存在一些非常重要的这种类型的关系，是我们无法表示的。例如，有了：

```
template<class T> class Ptr { // pointer to T
    // ...
};
```

我们可能常常希望把大家都已经习惯了的内部指针之间的继承关系也提供给这些用户定义的Ptr。例如：

```
void f(Ptr<Circle> pc)
{
    Ptr<Shape> ps = pc; // can this be made to work?
}
```

我们当然想允许这种东西，只要Shape确实是Circle的直接的或者间接的公有基类。特别是David Jordan代表一个面向对象数据库提供商的共同体，向标准化委员会提出了对这种灵巧指针性质的要求。

成员模板——目前还不是C++的组成部分——为此提供了另一种解决方案：

```
template<class T1> class Ptr { // pointer to T1
    // ...
    template<class T2> operator Ptr<T2> ();
};
```

我们需要定义一种转换操作，使得，当且仅当T1*可以赋值给T2*时，可以接受从Ptr<T1>到Ptr<T2>的转换。这一点可以通过为Ptr另外提供一个建构函数的方式做到：

```
template<class T> class Ptr { // pointer to T
    T* p;
public:
    Ptr(T*);
    template<class T2> operator Ptr<T2> () {
        return Ptr<T2>(p); // works iff p can be
                            // converted to a T2*
    }
    // ...
};
```

这种解决方案有一个极好的性质：它不使用任何强制。当且仅当p可以是Ptr<T2>的建构函数的参数时，这个返回语句能够被编译。现在p是T1*，而建构函数期望一个T2*参数。这是通过使用技术来增加限制（15.4.2节）的一个巧妙应用。如果你更喜欢将这种额外的建构函数保持为私用的，那么就可以使用Jonathan Shaprio所建议的技术：

```
template<class T> class Ptr { // pointer to T
    T* tp;
    Ptr(T*);
    friend template<class T2> class Ptr<T2>;
public:
    template<class T2> operator Ptr<T2> ();
    // ...
};
```

有关成员模板的讨论在15.9.3节里。

15.9.2 转换

与此密切相关的另一个问题是，不存在一种通用方法，以定义从一个模板类产生出的不同类之间的转换。例如，考虑下面的complex模板，它能够从各种标量类型出发定义出复数类型：

```
template<class scalar> class complex {
    scalar re, im;
public:
    // ...
};
```

有了这个东西，我们就可以使用complex<float>、complex<double>等等。不管怎样，在做这些事情时，我们会希望能做从一个具有较低精度的complex到具有较高精度complex的转换。例如：

```
complex<double> sqrt(complex<double>);

complex<float> c1(1.2f, 6.7f);
complex<double> c2 = sqrt(c1); // error, type mismatch:
                                // complex<double> expected
```

我们当然想有一种方法能使这里的sqrt调用合法化。这导致程序员放弃complex的模板定义途径，转而去使用重复的类定义：

```
class float_complex {
    float re, im;
public:
    // ...
};

class double_complex {
    double re, im;
public:
    double_complex(float_complex c) :re(c.re), im(c.im) {}
    // ...
};
```

采用这种重复定义就是为了通过建构函数定义有关的转换。

与前面类似，我能想到的所有解决办法都要求嵌套的模板和某种限制形式的组合。同样，实际的限制可以是隐含的：

```
template<class scalar> class complex {
    scalar re, im;
public:
    template<class T2> complex(const complex<T2>& c)
        : re(c.re), im(c.im) {}
    // ...
};
```

换句话说，你可以从一个complex<T2>构造出一个complex<T1>，当且仅当你可以用T2去参数化T1。这看起来也是很合理的。非常有趣的是，这种定义方式实际上覆盖了普通的复制建构函数。

这个定义就使上面sqrt()的例子变成合法的了。但不幸的是，这种定义方式也会允许各种complex值之间的窄转换，因为C++允许标量类型的窄转换。自然，对于complex的这个定义，如果一个实现能够对标量的窄转换做出警告的话，它也将自动地对complex值的窄转换产生警告。

我们可以通过typedef重新给出“传统的”名字：

```
typedef complex<float> float_complex;
typedef complex<double> double_complex;
typedef complex<long double> long_double_complex;
```

我个人认为不用typedef的形式更容易读。

15.9.3 成员模板

按照ARM的定义，C++里不允许将模板作为类成员。做这样的规定，惟一的原因是我无法在使自己满意的程度上确认这种嵌套不会成为严重的实现问题。成员模板出现在原始的模板定义里，我在原则上赞成所有作用域结构的嵌套（3.12节，17.4.5节），我也不怀疑成员模板一定会有用处，我从来也没有任何有关可疑的实现问题的确凿证据。万幸的是我犹豫了。如果我真的将成员模板接纳到C++里，不加任何限制，我就会在无意之中打破C++的对象布局模型，而后将不得不撤销该特征的一部分东西。请考虑下面这个看起来很有前途的想法，想作为双重发送（13.8.1节）的一个更优雅一些的变形：

```
class Shape {
    // ...
    template<class T>
        virtual Bool intersect(const T&) const =0;
};

class Rectangle : public Shape {
    // ...
    template<class T>
        virtual Bool intersect(const T& s) const;
};

template<class T>
virtual Bool Rectangle::intersect(const T& s) const
{
    return s.intersect(*this); // *this is a Rectangle:
                            // resolve on s
}
```

绝不能让这种东西合法化，否则，每次有人用一个新参数类型调用Shape::intersect()时，我们就必须给Shape的虚函数表再增加一个项。这将意味着，只有连接程序才能去构造虚函数表，并在表中设置有关的函数。因此，成员模板绝不能是虚的。

我只是到ARM出版以后才发现了这个问题，也因为将模板限制为只能定义在全局作用域里而得到了拯救。另一方面，如果没有成员模板，在15.9节提出的转换问题就没有办法解决。成员模板是1994年3月在圣迭戈的会议上被接受进C++的。

请注意，在许多情况下，显式描述模板函数的模板参数能作为替代嵌套模板类的另一种方式（15.6.2节）。

15.10 模板的实例化

C++里原来没有“实例化”模板的运算符[Stroustrup, 1988b][ARM]；也就是说，不存在这样的操作，通过它能够用一集特定的模板参数去显式地产生类声明或者函数定义。这样规定的原因是，只有到程序已经完成，才能知道到底需要对哪些模板进行实例化。许多模板是在库里定义的，而许多实例化是由用户直接或间接引起的，他们甚至根本不知道这些模板的存在。这样看起来，要求用户提出做实例化的请求似乎并不合理（例如，通过某种像Ada那样的“new”运算符的东西）。更糟的是，如果存在模板实例化的运算符，那么就必须能正确处理一些情况，在这些情况中，程序里互不相干的若干部分都要求以同样的模板参数集对同样的模板函数进行实例化。解决这个问题的方式必须能避免代码重复，同时也不能阻碍动态连接。

ARM里讨论了这个问题，但是没有给出一个确定的答案：

“这些规则意味着，究竟应该从函数模板定义生成哪些函数，这个决策只有到程序完成后才能做出，也就是说，在知道了哪些函数能用之前是不能做的。

如前所述，检查错误的问题已经被推迟到有可能做的最后时刻：在开始连接之后，模板函数的定义都已生成了的那一点。按照许多人的体验，这实在是太晚了。

如前所述，有关规则也将最大限度地依靠程序设计环境。找到模板类、模板函数、以及那些为产生这些模板函数所需要的类的定义，完全都是系统的责任。对于某些系统而言，这会使事情变得过于复杂而无法接受。

如果引进一种机制，允许程序员去说‘对于这些模板参数生成出这些模板函数’，上面这两个问题都可以得到缓解。对任何环境而言，这些都很容易做到，也能保证只要提出要求就可以去检查与某些特定模板函数定义有关的错误。

但是，这种机制究竟应该被看作是语言的一部分，还是程序设计环境的一部分，这个问题还不太清楚。看来还需要更多的经验，由于这个原因，这种机制一直属于环境——至少暂时是这样。

采用最简单的机制去保证模板函数定义的正确生成，这实际上是把问题留给了程序员。连接程序将告诉我们到底需要哪些定义，包含非在线模板函数定义的文件可以与关于需要使用哪些模板参数的指示一起进行编译。更复杂的系统可以在这种完全手工方式的基础之上构造起来。”

现在已经有了各种各样能够使用的实现。经验表明，这个问题至少是像原来所预期的那么困难，因此需要某些比现存实现更好的东西。

Cfront实现 [McCluskey, 1992] 完全按照原来的模板定义 [Stroustrup, 1988b][ARM] 自动做模板的实例化。简单说，在连接程序运行时，如果缺少了某些模板函数实例，它就会调用编译程序，从模板的源代码生成所缺的目标代码。这个过程反复进行，直到所有模板都完成了实例化。模板和参数类型定义根据文件命名规则找出来（如果需要）。在需要的地方，这些规则由用户提供的目录文件作为补充，这种文件把模板和类的名字映射到包含有关定义的文

件。编译程序有一种处理模板实例化的特殊模式。这种策略通常都工作得很好，但是在某些情况下也发现了三个很恼人的问题：

(1) 很糟的编译和连接执行性能：一旦连接程序确定了需要进行实例化，它就会调用编译程序去生成所需要的函数。这件事做完后，又需要重新调用连接程序去连接新的函数。某些系统里可能无法让编译程序和连接程序一直处在运行之中，这样就会带来很大开销。一种好的库机制有可能大大减少编译程序需要运行的次数。

(2) 与某些源代码控制系统之间很糟糕的交互关系：有些源代码控制系统对于源代码是什么，目标代码如何由这些源代码产生都有非常明确的看法。这种系统将不能与这里讲的编译系统很好地相互作用。因为这种编译过程是通过编译程序、连接程序和库的相互作用才产生出完整的程序（正如在(1)中给出的梗概）。这并不是语言的错误，但对于那些必须与上述这类源代码控制系统一起生活的程序员而言，这个情况就不是一种安慰了。

(3) 很难隐蔽实现的细节：如果我用模板实现了一个库，那么这些模板的代码就必须包含在库中，以便用户能连接我的库。需要这样做的原因是，只有到了最后连接的时候才能看到生成模板实例的需要。要想绕过这个问题，仅有的办法就是（设法）使生成的目标代码能包含我在实现中所使用的模板的每个版本。这必然会导致目标代码的膨胀，因为实现者要试着去覆盖所有可能的需要——而任何一个应用都将只会用到可能模板实例的一个小子集。还应该注意到，如果模板实现的实例化直接依赖于用户实际实例化了的那些模板的话，那么就必须采取延迟实例化的方式。

15.10.1 显式的实例化

看起来，缓和这些问题的最有前途的途径就是可选的显式实例化。这种机制可以通过额外的语言工具，或者通过依赖于实现的`#pragma`，或者通过语言内的某种指示符。所有这些方式都试验过，也取得了一些成功。在这些方式中我最不喜欢`#pragma`。如果在语言里需要有一种显式实例化的机制，我们应该要一种最普遍的具有完好定义的语义的东西。

如果有一个可选的实例化运算符，那么能获得哪些益处呢？

- 1) 用户将可能描述实例化的环境。
- 2) 用户将能以一种与实现相对无关的方式预先建立库的某些常用实例。
- 3) 这些预先建立的库将能与使用它们的那些程序的运行环境的改变无关（它们只依赖于实例化的上下文）。

下面将要描述的有关实例化请求的机制已经在San Jose会议中被采纳了，它起源于Erwin Unruh的一个建议书。语法形式被选择成与其他显式描述模板参数的形式相一致，就像使用类模板（15.3节），模板函数调用（15.6.2节），新的强制运算符（14.2.2节，14.3节），以及模板特殊化（15.10.3节）一样。实例化请求具有如下形式：

```
template class vector<int>;           // class
template int& vector<int>::operator[](int); // member
template int convert<int,double>(double); // function
```

关键字`template`又一次用到这里，因为不希望引进新关键字`instantiate`。模板声明与实例化请求是很容易区分的，模板定义的开始总是模板参数表`template<`，仅有`template`就表示是实例化请求。对函数用的是完全的描述形式，没有采用下面这种简写形式：

```
// not C++:

template vector<int>::operator[]; // member
template convert<int,double>; // function
```

这也是为了避免重载模板函数带来歧义性，为编译程序进行一致性检查提供冗余信息。因为实例化请求比强制出现得更少一些，强调记法紧凑的价值不大。还有，像模板函数调用一样，如果能从函数参数推导出模板参数，那么就可以省略掉它们（15.6.1节）。例如：

```
template int convert<int>(double); // function
```

当一个类模板被显式地实例化时，提供给编译程序的每个成员函数也都被实例化。这就意味着一个显式的实例化可以用来作为一种限制检查（15.4.2节）。

实例化请求机制可能显著地改进连接和重新编译的效率。我看到过这样的例子，如果将其中所有模板实例捆绑起来放进了一个编译单元，能将编译的时间从若干个小时缩短到同样数目的分钟。如果有一种手工的优化机制能有这样显著的提速，我将会愿意去接受它。

如果一个模板被显式地用同一集模板参数做了两次实例化，那么又会出现什么情况呢？这个问题相当棘手（很正当，按我的想法）。如果将它作为一种无条件的错误，显式实例化就可能成为一个严重障碍，阻碍从分别开发的部分出发去组合出整个程序。这也是原来不想引进显式实例化运算符的初始原因。在另一方面，想一般性地抑制多余的实例化又是非常困难的事情。

委员会决定对这个问题退后一步，给实现留下一些自由：将多重实例化作为一种并不要求做的诊断。这实际上是允许一个聪明的实现忽略掉冗余的实例化，以避免在从库出发组合程序时可能出现的各种问题，如果这些库已经采用了如上所述的实例化。但是，无论如何，这个规则并不要求一个实现必须是聪明的。“不太聪明的”实现的用户就必须去避免多重实例化，但是如果他们并没有这样做，可能出现的最坏情况就是他们的程序无法装载^Θ，不会出现不声不响地改变程序意义的事情。

如前所述，语言并不要求显式的实例化。显式实例化只不过是一种对编译和连接过程进行手工优化的机制。

15.10.2 实例化点

在模板定义中有一个最困难的方面，那就是精确地认定在模板定义中所使用的名字引用的到底是哪个定义。这种问题通常被称为“名字的约束问题”。

这一小节将描述经过修订的名字的约束规则，这是许多人最近几年的工作结果，特别应提到扩充工作组的成员，Andrew Koenig、Martin O'Riordan、Jonathan Shapiro和我。这些规则能够被接受（1993年11月，San Jose），也是得益于大量的实现方面的经验。

考虑下面的例子：

```
#include<iostream.h>
#include<vector.h>

void db(double);
```

^Θ 作者想说的是，因为出现过多的重复代码而导致结果程序极度膨胀，以至无法装入和运行。——译者注

```

template<class T> T sum(vector<T>& v)
{
    T t = 0;
    for (int i = 0; i < v.size(); i++) t = t + v[i];
    if (DEBUG) {
        cout << "sum is " << t << '\n';
        db(t);
        db(i);
    }
    return t;
}
// ...

#include<complex.h>                                // #2
void f(vector<complex>& v)
{
    complex c = sum(v);
}

```

原来的定义说，在模板里使用的名字都将在实例化的位置点上建立起约束，而实例化点恰在第一次使用这个模板的那个全局定义之前（上面的#2位置）。这样就至少出现了三个我们并不希望的性质：

(1) 在模板的定义点无法进行任何错误检查。例如，即使DEBUG在那一点无定义，也不能产生任何错误信息。

(2) 在模板定义之后定义的名字也可以被找到和使用。这经常（不一定总是）使阅读模板定义的人感到很吃惊。例如，人们通常会期望调用db(i)能被解析为在其前面定义的db(double)，但是如果在...处包含了一个db(int)声明，按照常规的重载解析规则，用的就应该是db(int)，而不是db(double)。另一方面，如果在complex.h里定义了一个db(complex)，我们就需要db(t)被解析为调用db(complex)，而不由于在模板定义里看不见合法的db(double)调用而被当作一个错误。

(3) 当sum被用于两个不同的编译单位时，在实例化点上可以使用的名字集合很可能不同。如果sum(vector<complex>&)因此而取得了两个不同的定义，结果生成的程序在惟一定义规则（2.5节）之下就是非法的。当然，在这些情况下，检查惟一定义规则已经超出了传统C++实现的范围。

进一步说，原来的规则并没有明确地概括另一种情况，即模板函数的定义并不包含在这个特定的编译单位里的情况。例如：

```

template<class T> T sum(vector<T>& v);

// ...

#include<complex.h>                                // #2
void f(vector<complex>& v)
{
    complex c = sum(v);
}

```

关于应该如何去找到sum()函数模板，有关规则没有给实现者或用户任何指导性的意见。因

此不同的实现者采用了不同的启发方式。

这里的一般性问题是，在一个模板实例化中，实际涉及到三个上下文，它们又无法清楚地相互分离：

- 1) 模板定义的上下文。
- 2) 参数类型声明的上下文。
- 3) 模板使用的上下文。

模板设计的全部目标就在于保证确实存在可用的充分的上下文信息，以便模板定义对它的实际参数真正有意义，而不会从调用点的环境中“随便”取来一些什么东西。

原设计完全依赖惟一定义规则（2.5节）来保证它合乎情理。这里采用的假定是，如果出现某些偶然的东西，影响到所产生函数的定义，那么这种事情不大可能一致地出现在使用模板函数的所有地方。这是一个很好的假定，但是——也是由于很好的理由——实现通常并不检查这种一致性。最终的效果就是只有合理的程序能够工作。但无论如何，希望模板实际上就是宏的人们又可以从这里又开去，去写出某些程序，使它们能以不适当的方式（按照我的观点）去利用调用的环境。还有，实现者如果想为一个模板定义集成起一个环境，以便提高实例化的速度，那么他们就又遇到了难题。

对定义点的定义加以精炼，要求它既要比原来的规则更好，又不打破合理的程序，这项工作是很困难的，但又是必须做的。

第一个意图是想要求所有在模板里使用的名字都必须在模板的定义点进行定义。这也能够使定义更容易读，保证不可能有任何不想要的东西被随便捡进来，并允许做早期的错误检查。可惜的是，这样做将阻止模板在本模板类的对象上进行操作。在上面例子里，+、f()和T的建构函数在模板的定义点都没有定义。我们将无法在模板里声明它们，因为无法描述它们的类型。例如，+可能是个内部运算符，或者是成员函数，或者是全局函数。如果它是个函数，它的参数可能具有类型T和T&等。这也是在描述参数限制（15.4节）时遇到的同一个问题。

既然模板的定义点和模板的使用点都不能为模板实例化提供足够好的上下文，我们必须找到一种解决方案，在其中组合这两者。

这个解决方案是把模板定义中使用的名字分成两大类：

- 依赖于模板参数的那些名字。
- 不依赖它们的那些名字。

第二类名字可以在模板定义的上下文里约束，而前者将在实例化的上下文中约束。只要我们能将其中的“依赖于模板参数”说清楚，这个概念就很清楚了。

1. 定义“依赖于T”

考虑“依赖于模板参数T”的定义时，作为第一个候选的是“T的成员”。当T是内部类型时，应该把内部的运算符看成是它的成员。可惜的是这样做还不够。考虑：

```
class complex {
    // ...
    friend complex operator+(complex,complex);
    complex(double);
};
```

要使这段代码能工作，至少必须把“依赖于模板参数T”扩展到包括T的友员。但是，即使这样做还是不够，因为一些重要的非成员函数不必是友员：

```

class complex {
    // ...
    complex operator+=(complex);
    complex(double);
};

complex operator+(complex a, complex b)
{
    complex r = a;
    return r+=b;
}

```

要求类的设计者提供将来所有使用模板的人可能需要的全部函数，这样说既不合理，也太受限制了。要做到百分之百的前瞻性几乎是不可能的。

因此，“依赖于模板参数T”就必须依据实例化点的上下文来确定，至少是在确定使用到T的全局函数方面。这样就不可避免地会出现用到了某些不希望的函数的可能性。但无论如何，这种可能性是非常小的。我们给了“依赖于模板参数T”一个最一般的定义，那就是说，一个函数调用依赖于某个模板参数，如果将这个实际模板参数从程序里去掉，这个调用的解析就会变化，或者就根本无法解析了。对于编译程序的检查而言，这个条件是直截了当的。调用依赖于参数类型T的例子如：

- 1) 该函数调用有一个形式参数，按照类型推导规则（15.6.1节），这个形参依赖于T。这方面的例子如f(T)、F(vector<T>)、f(const T*)等。
- 2) 按照类型推导规则，实际参数的类型依赖于T。这方面的例子如f(T(1))、f(t)、f(g(t))、f(&t)等。这里假定t的类型是T。
- 3) 一个调用的解析中使用了一个到类型T的转换，而按照1)和2)的定义，被调用函数的形式参数或者实际参数都不依赖于T类型。

最后一种情况的例子在实际代码中也可以看到。调用f(1)看起来不依赖T，它调用的函数f(B)也是如此，但是如果模板参数类型T有一个由int而来的建构函数，而它又能派生出B，因此f(1)的解析就将是f(B(T(1)))。

这三类依赖性已经囊括了我所见到过的所有例子。

2. 歧义性

如果查找#1（模板的定义点，在15.10.2节的例子里的#1点）和查找#2（使用点，在15.10.2节例子里的#2点）找到的函数不同，那又该怎么办？简单地说我们可以：

- 1) 偏向于查找#1。
- 2) 偏向于查找#2。
- 3) 认为这是个错误。

请注意，只有对非函数和那些所有参数类型在函数定义时都已知道的函数，才可能做查找#1；而对其他名字的查找都必须推迟到点#2进行。

本质上说，原来的规则就是“偏向于查找#2”，而这也就意味着使用常规的歧义性消解规则，因为只有在查找#2找到的是更好的匹配时，才可能是发现了一个与查找#1不同的函数。不幸的是，这样做将使你在读模板定义的正文时很难相信自己所看到的东西。例如：

```

double sqrt(double);

template<class T> void f(T t)

```

```

{
    double sq2 = sqrt(2);
    // ...
}

```

看起来sqrt(2)明显地是要调用sqrt(double)。但是，很可能在查找#2处确定的正好是sqrt(int)。这在大部分情况下可能都不是问题，因为“必须依赖于某个模板参数”的规则将能够保证sqrt(double)被使用，它是最明显的解析。但是，如果T本身就是int，那么调用sqrt(int)也将依赖于模板参数，这样，调用就应该被解析为sqrt(int)了。这是将查找#2考虑进来之后的一种不可避免的推论。无论如何，我认为它极其混乱，希望能避免这种东西。

而在另一方面，我又认为必须偏向于查找#2，因为，只有这样才能解决在把基类成员作为一种常规的类使用时可能出现的问题。例如：

```

void g();

template<class T> class X : public T {
    void f() { g(); }
    // ...
};

```

如果T有一个g()，那么g()应该按照非模板类匹配的方式被调用：

```

void g();

class T { public: void g(); }

class Y : public T {
    void f() { g(); } // calls T::g
    // ...
};

```

换句话说，在通常的“没有妨碍”的情况里，坚持查找#1找到的东西看来是正确的。这正是C++里面查找全局名字时所做的事，正是这种规矩使得大量的早期检查得以进行，允许大量的模板预编译。也正是它能够提供最大的保护，防止在写模板的人不知道的上下文中出现“偶然的”名字劫持。多年以来我已经意识到这些事情的重要性。有些实现者，特别是Bill Gibbons，为偏向查找#1做了很有说服力的辩护。

有一段时间，我更喜欢将两个查找中发现两个不同函数的情况作为一种错误，但是这样做会给实现者带来很大麻烦，又不能给用户带来显著的利益。此外，这样还可能使模板的某个使用上下文里的名字“打破”另一程序员写的模板代码，如果该程序员的意图就是想使用位于模板定义点的作用域中的名字，代码在其他方面都非常好。最后，经过在扩充工作组很长时间的工作以后，我改变了自己的看法。那些明显倾向于查找#1的论据实际上是一些非常技巧性的例子，很容易由写模板的人解决。例如：

```

double sqrt(double);

template<class T> void f(T t)
{
    // ...
    sqrt(2);           // resolve in lookup #1
}

```

```

sqrt(T(2)); // clearly depends on T
             // bind in lookup #2
// ...
}

```

和：

```

int g();

template<class T> class X : public T {
    void f()
    {
        g();      // resolve in lookup #1
        T::g();   // clearly depends on T
                  // bind in lookup #2
    }
    // ...
};

```

从本质上讲，这实际上是要求写模板的人做些事情，如果他真的希望去使用某些在模板定义处无法看到的函数的话，他就需要更明确地表达自己的意图。这样做看来是把负担放到了最合适的地方，并能产生正确的默认行为。

15.10.3 专门化

一个模板就是对任意的模板参数描述好了一个函数或者一个类的定义。例如：

```

template<class T> class Comparable {
    // ...
    int operator==(const T& a, const T& b) { return a==b; }
};

```

描述的是对每个T都用==运算符做元素的比较。可惜的是这样做太限制了一点。特别是对C语言中用char*代表的字符串，一般说最好是用strcmp()做比较。

在开始设计时我们就发现这种例子到处都是，而且，这些“特殊情况”常常对通用性是最本质的东西，或者是由于性能的原因而极其重要。C风格的字符串就是这类情况的最好实例。

因此，我的总结是，需要有一种能将模板“专门化”的机制。可以通过接受一般的重载而达到这个目的，或是通过某种更特殊的机制。我选择用一种特殊机制，因为从本质上说，这是要处理由于C的不规范而带来的不规范问题，也因为采用重载必然会引起保护主义者的怒吼。当时我是想尽量谨慎和保守，现在我已经认为这是个错误了。专门化，按照其本来的定义也就是重载的一种受限的不规则的形式，无法与语言的其他部分很好地配合。

一个模板类或者函数能够被“专门化”。例如，有了模板：

```

template<class T> class vector {
    // ...
    T& operator[](int i);
};

```

人们可以给出专门化的东西，作为分离的声明，例如，定义vector<char>和vector<complex>::operator[](int)：

```

class vector<char> {
    // ...
    char& operator[](int i);
};

complex& vector<complex>::operator[](int i) { /* ... */ }

```

这就使程序员能为自己所需要的类提供更专用的实现。这件事。无论从性能的观点，还是从与默认方式不同的语义的观点看，都特别重要。这种机制是生硬的，但也是极其有效的。

按照我原来的观点，这样的专门化最好是放到库里，只要需要就能够自动地使用，而无须程序员的干预。这已经被证明是一种代价昂贵而且是有问题的服务方式。专门化将给理解和实现带来许多问题，因为这样就无法确知一个模板对于一集特定的模板参数究竟意味着什么——甚至在仔细考察模板的定义之后——因为这个模板可能已经在某个另外的编译单元里做了专门化。例如：

```

template<class T> class X {
    T v;
public:
    T read() const { return v; }
    void write(int vv) { v=vv; }
};

void f(X<int> r)
{
    r.write(2);
    int i = r.read();
}

```

假定f()使用上面定义的成员函数是很合理的，但是却又无法保证。因为在另外的某个编译单元里，也可能将X<int>::write()定义为别的与此完全不同的东西。

专门化可以被认为是在C++的保护机制上打开了一个漏洞，因为一个专门化的成员函数可以访问模板类的私用成员函数，而且可能是以一种从阅读模板定义无法看出来的方式访问。这里还有更多的技术问题。

我的总结是，按照原来设计的那种专门化就像是一个德国鬼子，但同时也提供了具有本质意义的功能。我们怎样才能做到既提供了这种功能，又消灭了这个德国鬼子呢？经过许多很复杂的争论，我提出了一个特别简单的解决方案，这个方案在San Jose会议上获得通过：一个专门化必须在使用之前就已经声明。这就简单地将专门化放进类似常规重载规则的管理之下。如果在使用点的作用域里没有专门化，那么就采用一般的模板定义。例如：

```

template<class T> void sort(vector<T>& v)
/* ... */

void sort<char*>(vector<char*>& v); // specialization

void f(vector<char*>& v1, vector<String>& v2)
{
    sort(v1); // use specialization
    // sort(vector<char*>&)

    sort(v2); // use general template

```

```

        // sort(vector<T>&), T is String
}

void sort<String>(vector<String>& v); // error: specialize
                                         // after use

void sort<>(vector<double>& v); // fine: sort<double>
                                         // hasn't yet been used

```

我们曾考虑为请求专门化提供一个明显的关键字，例如：

```
specialise void sort(vector<String>&);
```

但是San Jose会议上的情绪是坚决反对新关键字。在这种国际性会议上，我也没有办法使人们在采用拼写specialize还是specialise的问题上达成一致意见^Θ。

15.10.4 查找模板定义

在传统上，C++程序和C程序一样都是由一集文件构成的。这些文件被组合进一个个编译单位，许多文件依据规定被编译连接进程序里。例如，.c文件是源文件，它们通过包含.h文件以获得程序其他部分的信息。编译程序从一些.c出发去生成目标文件，常常称为.o文件。程序的可执行版本就是简单地通过连接这些.o而得到的。档案（archive）和动态连接库使问题进一步复杂化了，但是并没有改变这个整体的画面。

模板并不能很好地放进这个画面里，这也正是与模板实现有关的许多问题的根源。一个模板不仅仅是一段源代码（通过模板实例化而产生的东西更像传统意义上的源代码），一些模板定义也不像是属于.c文件。从另一方面看，模板并不正好就是类型和界面信息，因此它们也不像是属于.h文件。

ARM并没有为实现者提供充分的指导（15.10节），这就导致出现了各种各样的实现模式，也变成了移植的障碍。有些实现要求把模板放进.h文件里。这样做会引起一些性能问题，因为需要为每个编译单位提供的信息太多，也因为每个编译单位都要依赖于它的.h文件里的所有模板。简单地说，模板函数定义并不属于头文件。另一些实现则要求将模板函数定义放在.c文件里。这也引起了在需要做实例化时如何找到模板函数定义的问题，它还使得很难组合出为实例化所用的上下文。

我想，对这些问题的任何合理解决办法都需要基于一种新认识，就是说，一个C++程序不仅仅是一集相互无关的分别编译单位，即使在编译期间这个观点也对。按某种方式，必须认识到这里有一个居于中心的概念，与模板以及其他事项有关的信息会影响多个编译单位。在这里，我将这个中心称为陈列台（repository），因为它的作用就是存放编译程序在处理程序的各独立部分时所需要的信息。

可以把陈列台想象成是一个始终存在的符号表，在其中每个模板对应一项，编译程序用它来维护声明、定义、专门化、使用等等的踪迹。给出了这个概念，我就可以勾勒出一个实例化模型，它能支持所有的语言功能，调解.h和.c文件的当前使用，不要求用户知道这个陈列台的任何东西，并为错误检查、优化和编译连接效率等等实现中必须考虑的问题另外提供一些可能方式。请注意，这是一个实例化系统的模型，而不是一条语言规则或者一种特定的

^Θ 作者的意思是在这里牵涉到英国英语与美国英语之争。——译者注

实现。仍然可能有多种不同的实现，但是我想用户可以忽略掉那些细节（在大部分时间），并按这种方式去考虑有关的系统。

让我从一个编译程序的观点来描绘有关的情况。通常，.c文件被送给编译程序，它们包含了一些使用.h文件的#include指令。编译程序只知道已经提供给它的代码，也就是说，它绝不到文件系统里去翻找，试图去发现还没有提供给它的模板定义。当然，编译程序需要用陈列台去“记录”已经见到过的模板，以及这些模板的出处。这个模式很容易扩充，以便包含通常的档案概念等等。下面是编译程序在一些关键点上做些什么样的简单描述：

- 遇到一个模板声明：这个模板现在可以用了。将这个模板放进陈列台。
- 在一个.c文件里遇到一个函数模板定义：尽可能地处理这个模板，并把它放进陈列台。如果它已经被放在那里了，除非现在的一个是同一模板的一个新版本，否则就给出一个重复定义的错误。
- 在一个.h文件里遇到一个模板的定义：尽可能地处理这个模板并把它放进陈列台。如果它已经被放在那里了，我们就检查那个已经放在里面的模板是否来自同一个头文件。如果不是就给出一个重复定义错。然后检查是否违反了惟一定义规则，即检查当前的定义是否与原来的定义一模一样。如果不一样，除非它是同一模板的一个新版本，否则就给出一个重复定义的错误。
- 遇到一个函数模板专门化声明：如果需要就给出一个专门化之前使用的错误。这个专门化现在可以使用了，将这个声明放进陈列台。
- 遇到一个函数模板专门化定义：如果需要就给出一个专门化之前使用的错误。这个专门化现在可以使用了，将这个定义放进陈列台。
- 遇到一个使用：将这个模板被以这集模板参数的方式使用的事实记入陈列台。在陈列台里查找是否已经有了一个一般的或一个专门化的模板定义。如果有，就可以进行错误检查和/或优化。如果该模板以前还没有以这一集模板参数的方式使用过，则可以生成对应的代码，也可以将代码生成推迟到连接时进行。
- 遇到一个显式的实例化请求：检查该模板是否已经有定义了；如果没有，给出一个模板无定义错误。检查是否有了一一个专门化的定义；如果有则给出一个实例化和专门化错误。检查这个模板是否已经用这一集模板参数做过实例化；如果是，可以给出一个重复实例化错误，或者也可以忽略掉这个实例化请求；如果不是，则可以生成代码，也可以将代码生成推迟到连接时。在这两种情况下，对每个模板类成员函数生成的代码都提供给编译程序。
- 程序被连接：为所有还没有生成代码的模板使用生成对应的代码。重复这个过程，直到所有的实例化都已经完成。对所有缺少的模板函数给出使用但没有定义错误。

对一个模板和一集模板参数的代码生成涉及到在15.10.2节里提出的查找#2。自然，还必须执行对非法使用和不能接受的重载的检查，如此等等。

在有关惟一定义规则和反对多重实例化规则的检查方面，一个实现可以做得严格些或放松些。在这里并没有必须要做的诊断，所以，确切的行为方式可以看成是实现的质量问题。

15.11 模板的作用

在早期的C++里没有模板，这对C++的使用方式产生了重要的负面影响。现在模板已经有

了，我们可能把哪些事情做得更好一些呢？

因为缺乏模板，在C++里就无法实现容器类，除非是广泛地使用强制，并且总通过通用基类的指针或者void*去操作各种对象。原则上说，这些现在都可以删去了。但是我想，无论如何，由于误导，将Smalltalk技术应用到C++里（14.2.3节）而引起的对继承的错误使用，来源于C语言的弱类型技术的过度使用，都是极难根除的。

在另一方面，我也希望能慢慢地摆脱掉涉及数组的许多不安全的实践。ANSI/ISO标准库里有dynarray模板类（8.5节），所以人们可以使用它，或者使用其他“家酿”的数组模板，以尽量减少使用不加检查的数组。人们经常批评C和C++不检查数组的边界。这种批评中的大多数是一种误导，因为人们忘记了，这只是说你能够使C数组出现越界错误，而不是说你必须这样做。数组模板使我们可以把低级的C数组贬黜到实现的内脏部分去，那里才真是它应该属于的地方。一旦C风格数组的使用频率降下去，它们的使用就将在类和模板的内部变得更加程式化，由于C数组而引起的错误也将急剧减少。在这些年里，这种情况已经在慢慢地发生，而模板正在加速这种趋势，特别是库里定义的模板。

模板的第三个方面在于它们为库的设计打开了许多全新的可能性，可以使派生和组合相结合（15.8节）。长远看这可能成为最重要的一个方面。

虽然支持模板的实现已经比较常见了，但它们也还不是广泛可用的^④。进一步说，大部分这样的实现还处在不大成熟的阶段。目前的这种情况也限制了模板对人们思考C++和程序设计时可能产生的影响。ANSI/ISO对各种黑暗角落的解决方案应该设法处理这两方面的问题，以使我们能看到模板在C++程序员的工具箱里取得中心地位，这也正是设计它的目标。

15.11.1 实现与界面的分离

模板机制完全是编译时和连接时的机制。模板机制的任何部分都不需要运行时支持。这当然是经过深思熟虑的，但也遗留下一个问题：如何让从模板产生的（实例化出来的）类和函数能够依靠那些只有到运行时才能知道的信息？与C++的其他地方一样，回答是使用虚函数。

许多人都表达了一种担心：模板好像过分地依靠了源代码的可用性。这被认为能带来两种副作用：

- 1) 你无法将自己的实现作为你的商业秘密。
- 2) 如果模板的实现改变了，用户的代码就必须重新编译。

这确实是大部分明显实现所遇到的情况，但是，利用提供界面的类派生出模板类的技巧可以限制这些问题的影响。模板经常被用来为某些可能是“秘密的”东西提供界面，以使那些东西可以修改，又不会对用户产生任何影响。15.5节的pvector是这方面的一个简单例子；13.2.2节中set例子的模板版本是另一个例子。我的观点是，关心这些事项的人应该用虚函数概念作为自己的另一种选择，我不需要再提供另一种跳转表^⑤。

④ 这是几年前的状况。今天所有较新的实现都提供了完整的模板实现，虽然许多实现还没有达到作者所希望的那种完美程度。——译者注

⑤ 跳转表，指代码地址表，用于实现间接跳入适当代码段的动作。虚函数是通过与类相关的虚表实现的，虚表就是一种跳转表。作者的意思很清楚，C++已经提供了虚函数，这是一种隐含的跳转表机制。如果你需要这种机制，就应该用它，不要再要求另一套本质上类似的机制。——译者注

同样可能创造出模板的一种半编译的形式，这将使实现者的秘密可以像目标代码一样安全——或者说是一样的不安全。

对于一些人来说，问题是如何保证对于需要保持秘密的模板，用户不能（无论直接还是间接地）通过实例化产生出新的版本。这可以通过不提供源代码的方式去保证。只要提供商能够预先通过实例化（15.10.1节）产生所有需要的版本，这个方法就可行。这些版本（也只有这些版本）被作为目标代码库发布。

15.11.2 灵活性和效率

由于模板的竞争矛头直接指向宏，对于它们的灵活性和效率的要求就非常严格。回过头看，有关工作的结果，得到的是一种没有抑制灵活性和效率的机制，也没有在静态类型检查方面做出任何让步。在用它去描述算法的时候，我偶尔地希望有高阶函数，但极少会想到运行时的类型检查。我怀疑，大部分通过限制或者约束“改进”模板的建议将会严重限制模板的用途，而又不能提供任何进一步的安全性、简单性或者效率作为补偿。下面引用的是Alex Stepanov在总结开发和使用一个重要的数据结构和算法库中所获得的经验：

“C++是一个足够强有力的语言——在我们的经验中第一个这样的语言——它使人能构造出类属的程序设计部件，在其中结合了数学的精确、美丽和抽象性，再加上非类属的手工编制代码的效率。”

我们仍然还没有完全发现将模板、抽象类和异常处理等等机制组合起来可能产生出的巨大能量。我并不认为在Booch的Ada和C++组件 [Booch, 1993b] 之间在规模上有十倍的差异是一个反常的例子（8.4.1节）。

15.11.3 对C++其他部分的影响

一个模板参数可以是一个内部类型，也可以是一个用户定义类型。这就产生了一种持续的压力，要求用户定义类型无论是在外观还是在行为上都要尽可能与内部类型相仿。可惜，用户定义类型和内部类型不可能做成具有完全一样的行为，因为无法既清除C语言内部类型的不规范性，而又不严重地影响与C语言的兼容性。在许多相对次要的方面，内部类型也从由模板带来的进步中有所收益。

在我第一次考虑模板问题以及后来使用它们的时候，都发现了一些情况，在其中内部类型的处理与类的处理之间有一些细微差别，这些也成为描述既能用于类参数，又能用于内部类型参数的模板的一种障碍。因此我开始工作，设法保证那些次要的语法和语义也能一致地应用于所有类型。这个努力一直继续到今天。

考虑：

```
vector v(10); // vector of 10 elements
```

对内部类型采用这种初始化形式是非法的。为了允许这种东西，我为内部类型也引进了建构函数和析构函数的概念。例如：

```
int a(1); // pre-2.1 error, now initializes a to 1
```

我考虑过进一步扩充这种概念，允许从内部类型出发做派生，允许为内部类型写内部运算符的显式声明。但是我撤回来了。

与采用int成员相比，允许从int出发做派生实际上不能给C++程序员提供任何重要的新东西。从本质上说，这是因为int根本没有可以由派生类覆盖的虚函数。更严重的是，C语言的类型转换规则实在是一塌糊涂，假装int、short等是行为良好的类，根本就行不通。它们可以或者是与C语言兼容的，或者是遵循C++对于类的相对良好的行为规范，但不可能同时满足两个方面。

允许重新定义内部的运算符，例如operator+(int,int)，就会使语言成为可变的东西。无论如何，如果能加进这样的函数，使人可以给它们传递指针值，或者用其他方式直接访问它们，看起来也是很诱人的。

从概念上说，内部类型确实也有建构函数和析构函数。例如：

```
template<class T> class X {
    T a;
    int i;
    complex c;
public:
    X() :a(T()), i(int()), c(complex()) { }
    // ...
};
```

X的建构函数对它的每个成员做初始化，方式是调用它们各自的默认建构函数。类型T的默认建构函数被定义为产生类型T的同一个值，这个值与类型T的没经过显式初始化的全局变量一样。这也是对ARM规定的一个改进，按那里的定义，X()将产生一个无定义值，除非X有默认的建构函数。

第16章 异常处理

不要慌！

——《去银河系的搭便车指南》

异常处理的目标——有关异常处理的假定——语法——异常的结组——资源管理——在建构函数里出错——唤醒与终止语义——非同步事件——多层异常传播——静态检查——实现问题——不变式

16.1 引言

在开始设计C++时就考虑了异常，但后来又推迟了，因为没有足够的时间去做彻底的工作，以探究有关的设计实现问题，也因为担心它们给实现带来复杂性（3.15节）。特别是那时就理解到某种很差的设计将会导致很大的运行时开销，并显著增加移植的代价。对于从多个分别设计的库出发去综合出程序，异常处理被认为是处理这种程序中错误的非常重要的机制。

C++异常机制的实际设计持续了许多年（1984~1989），它是在C++受到公众密切关注的情况下设计的第一个新的部分。除了数不清的黑板上的反复工作，正如每个C++特征都需要经历的那样，一些设计还被写成了文章，经过了广泛的讨论。Andrew Koenig深陷到最后的工作中，是几篇文章的合作者（与我一起）[Koenig, 1989a] [Koenig, 1990]。Andy和我在1987年11月去参加Santa Fe的USENIX C++会议的路上做出了最后模型之中最重要的部分。我还参加了在Apple、DEC（Spring Brook）、Microsoft、IBM（Almaden）、Sun以及其他一些地方的会议，做了有关设计草案的报告，并接到很有价值的反馈。特别是我找到了一些对提供异常处理机制系统有很多实践经验的人，弥补了我个人在这个领域的经验不足。我能记起来的第一次有关异常处理的严肃讨论是1983年在牛津，当时与来自Rutherford实验室的Tony Williams讨论的重点是容错系统的设计，以及在异常处理机制中静态检查的价值。

在ANSI C++委员会开始有关C++异常处理的争论时，有关C++异常处理的经验主要局限在基于库的实现上，包括Apple、Mike Miller[Miller, 1988]和其他一些工作；还有Mike Tiemann唯一的基于编译程序的实现[Tiemann, 1990]。这是很令人担心的，虽然当时大家已经有了相当广泛的共识：某种形式的异常处理机制对于C++是个好想法。特别是Dmitry Lenkov基于自己在HP的经验，表达了对异常处理的强烈愿望。对这个共识的一个值得注意的例外是Doug McIlroy，他认为，能使用异常处理将会使系统更不可靠，因为写库的人和程序员都会抛出异常，而不是想办法去理解和处理问题。只有时间才能告诉我们，Doug的预测在什么程度上将是真实的。自然，没有任何语言特征能防止程序员写出坏的代码。

像ARM里定义的那种异常处理的第一个实现是在1992年春天开始出现的。

16.2 异常处理的目标

在设计中做出了下面的假定：

- 异常基本上是为了处理错误。
- 与函数定义相比，异常处理器是很少的。
- 与函数调用相比，异常出现的频率低得多。
- 异常是语言层次上的概念——不仅是实现问题，也不是错误处理的策略。

这些说法以及下面列出的有关想法，都是从用来说明从1988年开始的设计演变的投影胶片上摘录下来的。

这里的意思是，异常处理

- 并不是想简单地作为另一种返回机制（有些人，特别是David Cheriton曾这样建议过），而是特别想作为一种支持构造容错系统的机制。
- 并不是想把每个函数都转变为一个容错的实体，而是想作为一种机制，通过它能给子系统提供很大程度上的容错能力，即使其中各个函数在写法上并没有关心全局的错误处理策略。
- 并不是想将设计者们都约束到一个“正确的”错误处理概念上，而是希望语言更有表达能力。

纵观这个设计努力的全过程，所有不同种类的系统的设计师们的影响越来越大，而从语言设计社团来的意见则逐渐减少。回过头看，对C++语言异常处理的设计产生影响最大的工作，是英格兰的Newcastle大学的Brian Randell和他的同事们从20世纪70年代开始，后来在许多地方继续进行的有关容错系统的工作。

下面这些有关C++的异常处理的思想逐渐发展起来了：

- 1) 允许从抛出点将任意数量的信息以类型安全的方式传递到异常处理器。
- 2) 对于不抛出异常的代码没有任何额外的（时间或空间）代价。
- 3) 保证所引发的每个异常都能被适当的处理器捕获。
- 4) 提供一种将异常结组的方式，使人不但可以写出捕获单个异常的处理器，还可以写出捕获一组异常的处理器。
- 5) 是一种按默认方式就能对多线程系统正确工作的机制。
- 6) 是一种能够与其他语言合作的机制，特别是与C语言合作。
- 7) 容易使用。
- 8) 容易实现。

这里的大部分目标都达到了，其他一些（如3)和8)后来认识到或者是代价太昂贵，或者是太受限制，因此只是接近达到。错误处理是一项很困难的工作，程序员在这方面需要得到帮助，我认为，C++的机制已经提供了所有可能的帮助。一个过分热心的语言设计者可能会提出更多的特征和/或限制，而它们实际上可能使容错系统的设计和实现进一步复杂化。

我关于容错系统的观点是：这种系统必须是多层次的，这帮助我抵御了要求各种“高级”特征的喧嚣。在一个系统里不可能有这样的独立单元，它能够从所有在它里面可能发生的错误，或者“外界”对它的可能侵害中恢复起来。在最极端的情况下，电源可能出故障，存储单元的值可能没有明显原因就改变了。

到了某一点，一个单元就必须放弃，而将进一步的清理工作留给“更高层次”的单元。例如，函数可能向调用处报告一个灾难性的失误；进程可能必须以非正常的方式结束，让其他进程去完成恢复工作；一个处理器可能要求其他处理器的帮助；一个完整的计算机系统可

能必须请求操作员的帮助。按照这种观点强调下面的东西就是有意义的：在每个层次上，只使用相对比较简单的异常处理特征去设计错误处理机制，这样做实际上也能工作。

要想提供一些功能，使一个单独的程序就能从所有错误中恢复，这完全是一种误导，也会使错误处理策略变得非常复杂，其本身就会成为新的错误根源。

16.3 语法

与往常一样，语法总能吸引到比其实际重要性更多的注意力。到最后我停止在一种相当罗嗦的语法形式上，其中用了三个关键字和许多花括号：

```
int f()
{
    try {           // start of try block
        return g();
    }
    catch (xxii) { // start of exception handler

        // we get here only if 'xxii' occurs
        error("g() goofed: xxii");
        return 22;
    }
}

int g()
{
    // ...
    if (something_wrong) throw xxii(); // throw exception
    // ...
}
```

这里的try关键字完全是多余的，那些花括号{}也是一样，除了真正需要在try块里或者异常处理器中使用多个语句的情况之外。例如，如果允许写下面的东西就会更简单：

```
int f()
{
    return g() catch (xxii) { // not C++
        error("g() goofed: xxii");
        return 22;
    };
}
```

但不管怎样，我发现要想解释清楚引进多余的东西只是为了提供所需支持，减少用户的困惑，则是一件很难的事情。因为C语言社团在传统上就厌恶关键字，我一直试图避免为了异常处理而增加三个新关键字，但是当时我拼凑起的每种少用关键字的模式看起来都过于精巧和/或含混。例如，我曾试着用catch表示抛出和捕捉两种操作，这完全可以做得符合逻辑，也具有内在的一致性，但我在给人们解释这种模式时却没有成功。

选择关键字throw，部分的原因是因为比它更鲜明的词，raise和signal，都已经被C语言的标准库函数使用了。

16.4 结组

经过与十来个不同的支持某种形式的异常处理的系统的几十个用户的讨论，我得出一个

结论，能够定义异常结组也是很关键的一种能力。例如，一个用户应该能捕捉“所有的I/O库异常”，而不必确切地知道其中到底包括哪些异常。如果没有结组功能，就必然要做某种迂回工作。例如人可以通过编码方式，将原本不同的异常变成某个异常所携带的数据，或者简单地在认为要捕捉这组异常的各处列出该组里所有异常的名字。但是，无论如何，大部分人（如果不是全部的话）的经验都说明，这些迂回方式最终都将成为维护中的问题。

Andrew和我首先试验了一种模式，基于异常对象的建构函数动态地构造结组。但是看来这种方式背离了C++其他部分的风格，许多人——包括Ted Goldstein和Peter Deutsch——都注意到，这种结组实际上与类分层等价。我们后来采纳了由ML获得灵感的一种模式，在其中抛出的是对象，通过声明接受这个类型的对象的处理器去捕捉它。按照C++正常的初始化规则，一个类型为B的处理器将能捕捉任何由B派生的类D的对象。例如：

```
class Matherr { };
class Overflow: public Matherr { };
class Underflow: public Matherr { };
class Zerodivide: public Matherr { };
// ...

void g() {
{
    try {
        f();
    }
    catch (Overflow) {
        // handle Overflow or anything derived from Overflow
    }
    catch (Matherr) {
        // handle any Matherr that is not Overflow
    }
}
}
```

后来又发现，多重继承（第12章）机制为用其他方式很难处理的分类问题提供了一种很优雅的解决途径。例如，人们可以像下面这样声明一个网络文件错误：

```
class network_file_err
: public network_err, public file_system_err { };
```

一个network_file_err类型的异常既能被考察网络错误的软件处理，也能被考察文件系统错误的软件处理。我认为是Dniel Weinreb第一次指出了这种应用。

16.5 资源管理

异常处理设计的核心点实际上是资源的管理。特别是如果一个函数掌握着某项资源，如果发生了异常情况，语言应该如何帮助用户保证在函数退出时能够正确地释放这项资源呢？考虑下面从 [2nd] 里取来的简单例子：

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn, "w"); // open file fn
```

```

// use f

fclose(f); // close file fn
}

```

这看起来似乎能行。但是如果什么东西正好在fopen()的调用之后和fclose()的调用之前出了毛病，一个异常就可能导致use_file()直接退出，而没有调用fclose()。请注意，在不支持异常处理的语言里也完全可能发生同样的问题。例如，对C语言标准库函数longjmp()的调用就会产生同样的效果。如果要写能够容错的系统，我们就必须解决这类问题。一种基本解决方案大致具有下面的样子：

```

void use_file(const char* fn)
{
    FILE* f = fopen(fn, "r"); // open file fn
    try {
        // use f
    }
    catch (...) { // catch all
        fclose(f); // close file fn
        throw; // re-throw
    }
    fclose(f); // close file fn
}

```

把所有使用文件的代码都包括在一个try块里，这样就能捕捉到所有的异常、关闭文件并重新抛出这个异常。

这种解决方案的缺点是有些啰嗦，冗长乏味，而且可能代价昂贵。进一步说，一个啰嗦而冗长乏味的解决方案总是很容易出错的，因为它会使程序员感到厌倦。我们可以提供一种特殊终结机制，以避免重复地写释放资源的代码（在这里是fclose()），这样就能把这种解决办法弄得稍微简洁一点。但是这样做并不能对解决其中根本性的问题起任何作用：有弹性的程序通常需要写比传统程序更特殊和更复杂的代码。

幸运的是，存在一个更优雅的解决方案。问题的一般形式大致具有下面的样子：

```

void use()
{
    // acquire resource 1
    // ...
    // acquire resource n

    // use resources

    // release resource n
    // ...
    // release resource 1
}

```

资源应该以它们被分配的相反顺序进行释放，在典型情况下这一点非常重要。这与局部对象通过建构函数创建，由析构函数销毁的行为方式极其相似。因此，我们可以将资源的请求和释放问题适当地用一个带有建构函数和析构函数的类的对象来处理。例如，我们可以定义一个类FilePtr，它在行为上很像FILE*：

```

class FilePtr {
    FILE* p;
public:
    FilePtr(const char* n, const char* a) { p = fopen(n,a); }
    FilePtr(FILE* pp) { p = pp; }
    ~FilePtr() { fclose(p); }

    operator FILE*() { return p; }
};

```

而后我们就可以构造FilePtr，或者给它一个FILE*，或者给它fopen()所需要的参数。在任何情况下，FilePtr都将在自己的作用域结束处被销毁，它的析构函数将关闭有关的文件。我们的程序收缩到了最小的程度：

```

void use_file(const char* fn)
{
    FilePtr f(fn,"r"); // open file fn
    // use f
} // file fn implicitly closed

```

无论函数是正常退出，还是由于抛出了异常，析构函数都将会被调用。

我把这种技术称作“资源获取就是初始化”。它还可以扩展到特殊构造的对象，这种技术可以用于处理在建构函数中出现错误时应该做什么的问题，这类问题采用其他方式是很难处理的，参见[Koenig, 1990]或[2nd]。

在建构函数里出错

对于某些人来说，异常处理最重要的方面就是作为一种一般性的机制，报告在建构函数里发生的错误。考虑FilePtr的建构函数，它并没有检查文件是否正常地打开。更细心的编码可能是：

```

FilePtr::FilePtr(const char* n, const char* a)
{
    if ((p = fopen(n,a)) == 0) {
        // oops! open failed - what now?
    }
}

```

如果没有异常处理机制，那么就没有报告出错的直接方法了，因为建构函数没有返回值。这将导致人们采用一些迂回的方式，例如将构造出的对象设置为某种错误状态，将返回值指示字存入某个约定好的变量等。令人吃惊的是这很少成为一个重要的实际问题。但无论如何，异常处理提供了一种具有普遍性的解决办法：

```

FilePtr::FilePtr(const char* n, const char* a)
{
    if ((p = fopen(n,a)) == 0) {
        // oops! open failed
        throw Open_failed(n,a);
    }
}

```

更重要的是，C++异常处理机制能保证部分构造起来的对象也能正确地销毁，也就是说，将

那些已经构造完成的子对象销毁，而那些还没有构造的子对象就不做。这就使写建构函数的人能集中注意力，去对那些检查到失误的对象做错误处理。进一步细节见 [2nd, 9.4.1节]。

16.6 唤醒与终止

在异常处理机制的设计期间，引起最大争议的问题是它究竟应该支持哪种语义，是终止语义还是唤醒语义。也就是说，异常处理器应不应该能够提出请求，从异常的抛出点重新唤醒程序的执行。例如，由于存储耗尽而调用的一个例程在找到某些存储之后返回，这是不是个好主意？让因为除零错误而调用的例程返回一个用户定义的值？让因为发现软盘驱动器空而被调用的例程在要求用户插入软盘而后返回？

我个人开始时的观点是：“为什么不呢？这些看起来都是很有用的特征。我真的能看到一些地方，在那里确实应该采用唤醒语义。”经过后来的四年，我学到了许多其他东西，因此，对C++的异常处理机制采纳了相反的观点，通常被称作终止模型。

有关唤醒对终止的主要辩论一直在ANSI C++委员会里进行着，委员会作为一个整体讨论了这个问题，讨论还在扩充工作委员会、晚间技术论坛、以及委员会的电子邮件表里进行。这个争论从1989年12月开始（是时ANSI C++委员会成立），一直延续到1990年11月。很自然，这个问题也成为C++社团最感兴趣的论题。在委员会里，Martin O'Riordan和Mike Miller是唤醒观点的最主要的倡导者和辩护士，而Andrew Koenig、Mike Vilot、Ted Goldstein、Dag Brück、Dmitry Lenkov和我通常是终止语义意见的保护人。我在大部分时间主要是作为扩充工作组主席的角色，引导着这个讨论。经过了一个很长的会议之后，在DEC、Sun、TI和IBM的代表给出了许多经验数据之后，扩充工作组投票以22对1通过了终止语义。随后，在ARM里描述的异常处理机制（它遵循终止语义）在整个委员会里以33比4被接受。

经过1990年7月西雅图会议长时间的辩论之后，我将唤醒语义的论点做了如下的总结：

- 更通用（功能强，包含了终止语义）。
- 能统一起类似的概念与实现。
- 对于更复杂、非常动态的系统（如OS/2）是根本性的。
- 对实现而言并没有明显更大的复杂性和代价。
- 如果没有，你必须设法去伪造它。
- 为资源耗尽问题提供了一种简单的解决方案。

也类似地对支持终止语义的论点做了总结：

- 更简单、更清晰、更廉价。
- 导致更易管理的系统。
- 对所有东西都足够强有力。
- 能避免可怕的编码诡计。
- 对唤醒语义的一些重要的负面经验。

这两个意见表将争论过分简单化了，实际上这些争论是非常技术性的，彻底的。争论有时会变得异常激烈，要求减少约束的辩护士们表达了这样的观点，说终止语义的拥护者们想要给他们强加上一些很随意很限制人的程序设计观点。很清楚，终止与唤醒问题触到了有关软件应该如何设计的深层问题。争论从来都不是在两个势均力敌的小组间进行的，在每次论坛中，终止语义的拥护者通常都有4比1或者更大的多数。

有关唤醒方式的反复出现且最有说服力的论点包括：

- 1) 因为唤醒是比终止更一般的机制，应该接受它，即使对于其有用性有所怀疑。
- 2) 实际存在一些重要的情况，在那里程序因为缺乏资源而被卡住（例如存储耗尽、空的软盘驱动器等）。对于这些情况，唤醒方式使例行程序可以抛出一个异常，由异常处理器提供所欠缺的资源，而后再唤醒执行，就像根本没有发生异常一样。

有关终止方式反复出现且最确凿（对我而言）的论据是：

- 1) 终止比唤醒简单得多。事实上，唤醒将要求那些为继续点（continuation）和嵌套函数所用的关键性机制，而又不能提供这些机制的实际效益。
- 2) 有关唤醒所提出的论据2)中资源耗尽的处理方法本身就很不好，它将导致库与用户之间过于紧密的约束，容易引起错误，也很难理解。
- 3) 在许多应用领域里实际使用的主要系统都是采用终止语义写出的，因此，唤醒语义并不是必需的东西。

这最后一点还得到了理论论据的支持，Flaviu Cristian证明，有了终止语义之后唤醒就不必要了[Cristian, 1989]。

经过了几年的讨论，给我留下最深刻印象的是，一个人可以站在任何位置，编制出一套能使人信服的逻辑论据。在论述异常处理的开创性论文[Goodenough, 1975]里就是这样做的。我们现在是站在古希腊哲学家的位置，在争论着宇宙的本质，如此激烈而敏锐，以至根本就忘记了去研究它。因此，我一直要求那些在大系统上有实际经验的人们，请在来时提供数据。在唤醒方面，Martin O'Riordan报告说“Microsoft对于唤醒式异常处理有了若干年的正面经验。”但是缺乏特殊例子。此外，对于把OS/2版本1作为技术有效性证明的价值的怀疑也减弱了他的论点的力量。在PL/I中ON-条件方面的经验也被提出来，作为对唤醒方式支持和反对两方面的论据。

再后来，在1991年11月Palo Alto会议上，我们听到Jim Mitchell（来自Sun，以前在Xerox PARC）基于他个人的经验和数据为终止语义所做的绝妙的总结。Jim在过去20年间在好几个语言里使用异常处理，他也是唤醒语义最早的支持者，是Xerox的Cedar/Mesa系统的主要设计师和实现者。他的信息是：

“终止比唤醒更好，这不是一种观点的问题，而是许多年的经验。唤醒是非常诱人的，但却是站不住脚的。”

他用来自几个操作系统的经验支持自己的论断。最关键的例子是Cedar/Mesa：这个系统原来是由喜欢并使用唤醒的人们写出的，但是经过十年使用，在大约50万行的系统里只留下了一个唤醒——做上下文的询问。由于这个上下文询问实际上并不需要采用唤醒，他们删除了它并发现系统里这部分的速度显著提高了。以前使用唤醒的每个地方——在这十年期间——都变成了问题，而后被用更适合的设计所取代。简单说，唤醒的每个使用都表现了一个失误，没有保持相互分离的层次间抽象的不相交性。

Mary Fontana从TI的Explore系统中得到类似的数据，在那里发现唤醒语义只在排错中使用。Aron Insinga提供了另一些证据，说明在DEC的VMS里，唤醒只有非常有限的很不重要的应用。Kim Knuttila关于IBM两个大的长期项目给出了与Jim Mitchell如出一辙的故事。Dag Bruck转告我们的基于在L.M.Ericsson的经验也偏向于终止语义，这些又增强了我们的信念。

这样，C++委员会采纳了终止语义。

迂回地实现唤醒

很明显，唤醒中最重要的利益可以通过组合起函数调用和（终止的）异常而获得。考虑一个函数，用户想调用它，以获得某种资源X：

```
X* grab_X() // acquire resource X
{
    for (;;) {
        if (can_acquire_an_X) {
            // ...
            return some_X;
        }

        // oops! can't acquire an X, try to recover:
        grab_X_failed();
    }
}
```

尽可能地使所请求的X能够使用正是grab_X_failed()的工作，如果它无法做到这一点，那么就抛出一个异常；

```
void grab_X_failed()
{
    if (can_make_X_available) { // recovery
        // make X available
        return;
    }

    throw Cannot_get_X; // give up
}
```

这个技术正是对存储耗尽的new_handler方法（10.6节）的推广。当然，这种技术还存在着许多变形。我喜欢的是在某个地方用一个指向函数的指针，以便使用户可以“定制”有关的恢复机制。这种机制不会给系统带来实现唤醒所牵涉到的复杂性。一般说，它也不会给系统结构带来负面影响，不像一般性的唤醒机制那样。

16.7 非同步事件

C++异常处理机制明显无法直接处理非同步事件：

“异常能够用于处理信号那类的东西吗？在大部分C环境里，几乎可以肯定说不行。麻烦在于C所使用的一些函数不是可重入的，如malloc。如果中断出现在malloc里面并导致了一个异常，那么将无法防止异常处理器重新执行malloc。”

如果在一个C++实现里，调用序列及整个运行库都是围绕着可重入的要求设计的，那么就能允许信号抛出一个异常。在这种实现变成常见的东西之前（如果有那一天的话），从语言的观点上看，我们必须建议严格地分离异常和信号。在许多情况下按下面方式让信号与异常交互是合理的：让信号将信息存在一旁，而常规地由某个函数检查（登记）这里，该函数可能根据信号存储的信息转去抛出适当的异常 [Koenig, 1990]。”

我的观点恰好也反映了C/C++社团里关心异常处理的人们中大多数的观点。这个观点就是：

为了做出可靠的系统，你需要尽快地把非同步事件映射到某种形式的进程模型里。由于异常可能发生在执行中的任意一点，再者说，把对于一个异常的处理停下来，转去处理另一个与之无关的异常实际上就是要求混乱。低级中断系统应该尽可能与普通程序分离。

这种观点就排除了直接使用异常去表达某些东西，如按压DEL键；或者是用异常去取代UNIX里的信号。在这些情况中，某个低级中断例程必须按某种方式完成自己最少的工作，并尽可能将事件映射到某种东西，而这种东西能够在程序执行中的某个定义良好的点上激发一个异常。请注意，按照C语言的定义，信号不能去调用函数，因为在处理信号期间不能保证机器状态具有某种一致性，因此无法处理函数的调用和返回问题。

与此类似，其他低级事件，如算术溢出和除以零，也假定是由专用的低级机制处理，而不该由异常来处理。这就使C++在做算术时的行为方式也能与其他语言匹配。这样也避免了在超流水线体系结构中出现问题，在那里除以零这样的错误也是非同步的。要求对除以零等事件进行同步，这未必在所有机器上都能做到。而在那些能够做到这一点的地方，也需要刷新流水线，以保证这种事件在其他与之无关的计算开始前被捕捉到，这样会大大减慢机器速度（通常要差几个数量级）。

16.8 多层传播

存在着一些很好的理由，要求在一个函数里发生的异常只能隐式地传播到它的直接调用处。但C++里没有采用这种选择：

- 1) 现存成百万的C++函数，期望去修改它们以便传播和处理异常是不合理的。
- 2) 把每个函数做成一个防火墙并不是一种好想法，最好的错误处理策略是在其中指定一些主要界面，让它们关注非局部的错误处理问题。
- 3) 在混合语言环境里，不可能要求某个函数一定能具有某种活动，因为它完全可能是用另一种语言写的。特别是一个抛出异常的C++函数可能是由一个C函数调用的，而这个C函数转而又是由另一个打算捕捉异常的C++函数调用的。

第一个理由完全是实用性的，另两个则是根本性的：2)讨论的是系统的设计策略；而3)则是关于C++可能在什么环境里工作的论断。

16.9 静态检查

由于允许异常的多层传播，C++就丧失了一方面的静态检查。你无法简单地看一个函数就确定它可能抛出什么异常。事实上它可能抛出任何异常，即使在这个函数的体里连一个throw语句也没有。因为被它调用的函数可能做这种抛出。

一些人对此非常不满，特别是Mike Powell，他们试着去弄清楚C++的异常究竟能提供多强的保证。在理想的情况下，我们希望保证每个被抛出的异常都能由用户提供的适当处理器捕获。一般说，我们希望保证，只有那些明显地列在某个表里的异常可以逃出一个函数。C++提供了为描述一个函数可能抛出的异常所用的列表机制，这是由Mike Powell、Mike Tiemann和我在1989年的某个时候在Sun的一块黑板上开始设计的。

“在效果上，写下面的东西：

```
void f() throw (e1, e2)
{
    // stuff
}
```

等价于写：

```
void f()
{
    try {
        // stuff
    }
    catch (e1) {
        throw; // re-throw
    }
    catch (e2) {
        throw; // re-throw
    }
    catch (...) {
        unexpected();
    }
}
```

明确地声明函数可能抛出的异常，与在代码中直接进行与之等价的检查相比，其优点不仅在于节省了类型检查。最重要的优点是，函数声明属于用户可以看见的界面。而在另一方面函数定义并不是一般可见的，即使我们可以直接看到所有库的源代码，我们也强烈地希望不要经常地去看它。

另一个优点，就在于它使在编译时检查出许多未捕捉异常的错误有了实际的可能性 [Koenig, 1990]。”

理想的情况是，异常刻画应该在编译时进行检查，但是这就要求每个函数都必须与这个模式合作，而这又是不可行的。进一步说，这种静态检查很容易变成许多重新编译的根源。更糟糕的是，这种重新编译只有在用户掌握着所有需要重新编译的源代码时才可能进行：

“例如，如果被一个函数（直接或间接地）调用的另一个函数修改了自己所捕捉或抛出的异常，那么这个函数就可能需要修改和重新编译。这将导致那些（部分）使用从多个来源的库组合起来而生产的软件产品大大地拖延时间。因为这些库必须在所采用的异常方面达成事实上的一致意见。例如，如果子系统X处理从子系统Y来的一个异常，而Y的提供商为Y引进了一类新的异常，那么就必须修改X的代码去迎合这种情况。而X和Y的用户将无法升级到Y的新版本，要等到X的修改完成。在使用了许多子系统的地方，这种情况会造成瀑布式的延迟。即使是在不存在‘多供应商’问题的地方，这个情况也可能引起瀑布式的代码修改和大量的重新编译。

这种问题将会导致人们避免使用异常描述机制，或是去颠覆它 [Koenig, 1990]。”

这样，我们就决定只支持实时检查，将静态检查的问题留给另外的工具去做。

“在使用动态检查时也出现了一个等价问题。在这些情况下，无论如何，这个问题可以通过16.4节讨论的异常结组机制解决。异常处理机制的一种朴素使用就是把子系统Y新增加的异常都留在那里不予处理，或者通过某个明显的调用界面转换为一个对unexpected()的调用。

定义得更好的子系统Y应该将其所有的异常都定义为由一个类Yexception派生的。例如：

```
class newYexception : public Yexception { /* ... */ };
```

这就意味着如下声明的函数

```
void f() throw (Xexception, Yexception, IOException);
```

将能处理一个newYexception，方法就是将它传递给f()的调用者。”

进一步的讨论见[2nd]的第9章。

在1995年我们发现了一种模型，它能够允许对一些异常描述做静态检查并改进代码生成，而同时又不会引起上面所说的问题。因此，现在已经对异常描述做检查了，这也使函数指针的赋值、初始化和虚函数重载等不会引起类型违规问题。某些未预期的异常还是可能发生的，它们像以往一样在运行时进行捕捉。

实现问题

效率仍然是最重要的担心。很明显，人完全可以这样设计异常处理机制，使它的实现在线数调用序列方面产生显著的直接代价，或者由于需要防止可能出现的异常而在优化方面付出间接的代价。看起来我们成功地处理了这些担心，至少在理论上，C++异常处理机制可以这样实现，使不抛出异常的程序不付出任何时间代价。可以把实现安排成将所有的运行时间代价都集中到异常抛出的时候[Koenig, 1990]。也可能限制空间的开销。但却很难同时避免运行时间开销和代码规模的增大。一些实现现在已经能支持异常了，所以这种折中将变得更清楚。有关例子见[Cameron, 1992]。

很奇妙，异常处理并没有在任何实际程度上影响对象的布局模型。在运行时表示类型是必需的，以便能在抛出点和处理器之间进行通讯。但无论如何，这件事可以通过专用的机制完成，它不会影响到普通的对象。换一种方式，也可以利用支持运行时类型识别(14.2.6节)的数据结构。更关键的是，保持每个自动对象的确切生存时间轨迹现在变成了极重要的问题。直截了当地实现这种机制可能导致某种代码肿胀，即使实际被执行的附加指令数目很少。

我的理想实现技术是根据人们对Clu和Modula-2+实现所做的工作[Rovner, 1986]导出的。最基本的思想是放置一个代码地址范围的表，将计算状态和与之相关的异常处理对应起来。对其中的每个范围，记录所有需要调用的析构函数和可能调用的异常处理器。当某个异常被抛出时，异常处理机构将程序计数器与范围表中的地址做比较。如果发现程序计数器位于表中某个范围里，就去执行有关的动作；否则就解脱一层堆栈，使程序计数器退到调用程序中，再去查找范围表。

16.10 不变式

作为一种新的、正在演化中的然而又是大量使用的语言，C++吸引来了比平均份额更多的改进和扩充建议。特别是任何语言中的在某种意义上时髦的任何特征最终都会提到C++里来。Bertrand Meyer将前条件和后条件的传统想法更大众化了，在Eiffel里对它提供了直接的语言支持[Meyer, 1988]。人们很自然地也建议C++提供直接的支持。

C社团中的一部分人一直广泛地依靠assert()宏，但是在运行中却没有好的办法报告出现违背断言的情况。异常提供了处理这个问题的一种方式，而模板提供了一种避免依赖于宏

的途径。例如，你可以写一个Assert()模板，用它去模仿C语言的assert()宏：

```
template<class T, class X> inline void Assert(T expr, X x)
{
    if (!NDEBUG)
        if (!expr) throw x;
}
```

如果expr是假而且我们没有通过设置NDEBUG关闭检查，它就会抛出一个异常x。例如：

```
class Bad_f_arg { };

void f(String& s, int i)
{
    Assert(0<=i && i<s.size(), Bad_f_arg());
    // ...
}
```

这是这种技术的一个最不结构化的变形。我个人更喜欢将类的不变式定义为成员函数，而不是直接使用断言。例如：

```
void String::check()
{
    Assert(p
        && 0<=sz
        && sz<TOO_LARGE
        && p[sz-1]==0, Invariant);
}
```

由于很容易在现有的C++语言里定义、使用断言和不变式，这就使要求为特别支持程序验证特征而扩充语言的吵嚷减到了最小。随之，针对这些技术的大部分努力现在都转到标准化的建议方面[Gautron, 1992]，或者更野心勃勃的验证系统方面 [Lea, 1990]，或者就是简单地在现有框架中使用。

第17章 名字空间

总在下一个比事物大的上下文里
考虑该事物的设计问题。

——Eliel Saarinen

全局作用域的问题——解决方案的思想——名字空间，使用声明，以及使用指引
——如何使用名字空间——名字空间与类——与C语言的兼容性

17.1 引言

对所有不适合放进某个函数、某个**struct**或者某个编译单位的名字，C语言提供了一个统一的全局性的名字空间。这就带来了名字的冲突问题。我第一次与这个问题搏斗是在开始设计C++时，那时是想让所有名字都默认地属于编译单位，而要求用显式的**extern**声明使名字变成在其他编译单位里可以看见。正如在3.2节所说的，这个想法既不足以解决问题，又无法提供能够接受的兼容性，它失败了。

在我设计类型安全的连接机制时（11.3节），又重新考虑了这个问题。我注意到，对

```
extern "C" { /* ... */ }
```

的语法、语义和实现技术稍微做些修改，就能允许我们用

```
extern XXX { /* ... */ }
```

表示在XXX里声明的名字位于一个分离的作用域XXX里，要从其他作用域中访问就需要用量化的XXX::形式，和在类之外访问静态类成员完全一样。

由于一些原因，最主要就是缺少时间，这个想法休眠在那里，直到1991年早期，在一次ANSI/ISO委员会讨论中它又重新浮出水面。首先，来自Microsoft的Keith Rowe提出了一个建议书，建议采用记法：

```
bundle XXX { /* ... */ };
```

作为定义名字作用域的机制，用运算符use把在一个bundle里的所有名字引进另一个作用域。这引起扩充工作组的几个成员之间一场不那么热烈的讨论，包括Steve Dovich、Dag Brück、Martin O'Roedan和我。最后，来自Siemens的Volker Bauche、Roland Hartinger和Erwin Unruh将讨论中的思想加以精炼，提出了一个不使用关键字的建议：

```
:: XXX :: { /* ... */ };
```

这在扩充工作组内引起了一场更严肃的讨论。特别是Martin O'Riordan演示了这种::记法将导致歧义，因为::既用于类成员，也用于全局性的名字。

到1993年早期，借助于成兆字节的交换电子邮件和标准化委员会的讨论，我综合出一个具体的建议。按我的回忆，对名字空间做出技术贡献的有Dag Brück，John Bruns，Steve Dovich，Bill Gibbons，Philippe Gautron，Tony Hansen，Peter Juhl，Andrew Koenig，Eric Krohn，Doug

McIlroy, Richard Minner, Martin O'Riordan, John Skaller, Jerry Schwarz, Mark Terrible和Mike Vilot。此外, Mike Vilot论证说应该立即将这个想法开发成一个确定性的建议, 使这种机制能够用于处理ISO C++库中不可避免的名字问题。名字空间是在1993年7月的慕尼黑会议上通过加入C++的, 在1993年11月San Jose会议上, 决定用名字空间来控制标准C库和C++库的名字问题。

17.2 问题

如果只有一个名字空间, 要写出程序片段, 使它们可以连接到一起而无须害怕出现名字冲突, 实际存在着一些不必要的困难。例如:

```
// my.h:
char f(char);
int f(int);
class String { /* ... */ };

// your.h:
char f(char);
double f(double);
class String { /* ... */ };
```

对于这些定义, 第三方将很难同时使用my.h和your.h。

请注意, 这些名字中的一些将最终出现在目标代码里, 而某些程序将以不带源代码的方式销售。这就意味着, 不在程序中实际修改名字, 仅通过宏定义方式改变程序的表现, 并将这种东西提供给连接程序的方式根本就行不通。

迂回方法

也有一些迂回的解决方案。例如:

```
// my.h:
char my_f(char);
int my_f(int);
class my_String { /* ... */ };

// your.h:
char yo_f(char);
double yo_f(double);
class yo_String { /* ... */ };
```

这种方式很常见, 但它也很难看。除非作为前缀的串很短, 用户是不会喜欢它们的。另一个问题是, 实际上只有几百个两字符的前缀, 而现在已经有了成百的C++库。这是书中最常见的一个最老的问题。老的C程序员还应该记得那个年代, 那时需要给struct成员名一个或两个字符的前缀, 以防与其他struct成员的名字发生冲突。

宏机制可能将这种东西弄得更污浊 (或者更美妙, 如果碰巧你喜欢宏的话):

```
// my.h:
#define my(X) myprefix_##X

char my(f)(char);
int my(f)(int);
```

```

class my(String) { /* ... */ };

// your.h:
#define yo(X) your_##X

char yo(f)(char);
double yo(f)(double);
class yo(String) { /* ... */ };

```

这里的想法就是让连接时的名字带有很长的前缀，而在程序里仍然使用短名字。与其他所有宏模式一样，这种东西将给工具带来问题：或者是要求工具记录所有映射的轨迹（使工具更复杂），或者用户必须自己做（使程序设计和维护的工作更加复杂）。

另一种替代方式——受到那些不喜欢宏的人们的偏爱——就是把相关的信息都包裹进一个类里：

```

// my.h:
class My {
public:
    static char f(char);
    static int f(int);
    class String { /* ... */ };
};

// your.h:
class Your {
public:
    static char f(char);
    static double f(double);
    class String { /* ... */ };
};

```

不幸的是，这种方式也受到了许多小小的不方便的困扰。并不是所有全局性的声明都能很简
单地转变为一个类，如果你真的这样做了，有些东西的意义就可能发生变化。例如，为了避免
改变语义，全局函数和变量必须声明为static，函数体和初始化一般都必须与它们的声明
分离开，等等。

17.3 解决方案的思想

许多机制都可以用来作为名字空间问题的解决方案。确实，大部分语言都可以说至少有
某种基本的东西。例如，C有它的静态函数，Pascal有它的嵌套作用域，C++有它的类，但是
我们需要的是走向如PL/1、Ada、Modula-2、Modula-3 [Nelson, 1991]、ML [Wikström,
1987] 及CLOS [Kiczales, 1992] 一类语言，提供一个完整的解决方案。

那么，C++里应该怎样给我们提供一个好的名字空间机制呢？经过ANSI/ISO委员会的扩
充工作组邮件表里很长的内容丰富的讨论，提出了如下的一些东西：

- 1) 能够在连接两个库时不出现名字的冲突。
- 2) 能够在引进新名字时不必担心与别人已经有的名字冲突（例如，在我没听说过的某
个库中可能使用的名字，或者在某个我认为自己已经知道的库里的没有听说过的名
字等）。
- 3) 能够在库的实现里增加一个名字而又不影响库的用户。

- 4) 能够从两个不同的库里选择名字, 即使这两个库恰好用了同样的名字。
- 5) 能够不修改有关的函数本身, 而消解掉出现的名字冲突 (也就是说, 通过声明去操纵名字空间的解析方式)。
- 6) 能够在一个名字空间里增加新名字, 而不必担心这样做会导致那些用到其他名字空间的代码不动声色地改变意义 (我们无法对使用新加的名字空间的代码提供这种保证)。
- 7) 能够避免名字空间的名字之间发生冲突 (特别是使它们能具有比用户代码中所使用的名字更长的“真正”名字或者连接名字)。
- 8) 使名字空间机制能有处理标准库的能力。
- 9) 与C和C++的兼容性。
- 10) 不给名字空间的用户增加任何连接时或者运行时的额外成本。
- 11) 不给使用名字空间的用户增加比使用全局名字的用户更多的描述上的麻烦。
- 12) 使人能够在使用名字的代码里显式指明某个名字的假定出处。

此外, 解决方案必须是简单的。我把简单定义为:

- 1) 一种机制, 它能够在十分钟之内给出一个解释, 达到认真地使用所需要知道的程度。要将任何语言特征解释到语言专家满意的程度都需要更长的时间。
- 2) 某种C++实现者用不了两个星期就能够实现的东西。

很自然, 这种意义下的简单性无法严格地证明。例如, 理解某种东西所需要的时间可能因人而异, 因为其知识背景和能力水平的不同而产生很大的差异。

人们还曾经提出过一些性质, 我们有意将其排除在名字空间机制的评价准则之外:

- 1) 有能力将存在连接名冲突的二进制文件连接到一起。在任何系统里, 利用工具都可以做到这件事, 但是我没有看到任何语言特征能够很容易地在所有系统上实现, 而不需要花费大量的精力和开销。我们周围存在着太多的连接程序, 太多的目标代码格式, 改变它们并不可行。一个解决方案要想对C++有用, 它就应该只使用目前几乎所有的连接程序都已经提供的能力。
- 2) 为库里所用的名字提供任意同义词的能力。现成的机制, 如`typedef`、引用以及宏只是对一些特殊情况能够提供同义词, 而我根本就不信任一般性的重命名机制, 见12.8节。

这也就意味着, 非歧义性必须由程序片段的提供者编译到目标代码里。特别是库的提供商必须采用一种能允许用户消除歧义性的技术。幸运的是, 库的提供商也将从系统化地使用名字空间中获得最主要的利益, 因为他们 (部分地是通过他们的用户) 是当前状况的最主要受害者。

自然, 还可能给这些列表增加一些准则, 而且也不会有两个人对所有准则的重要性有完全相同的认识。但无论如何, 这些条目对于问题的复杂性和解决方案, 以及必须满足的要求给出了一个基本想法。

在第一次提出了这些准则之后, 我就有了机会去依据这些准则, 检查名字空间设计的简单性的情况。Peter Juhl用五天完成了一个指导性的实现, 我将名字空间的基本情况解释给一些人, 只用了几张投影片, 不到十分钟的时间。他们随后提出的问题显示出已有的理解, 而且还推出了一些我并没有解释的名字空间应用。这使我非常满意, 名字空间功能确实足够简单。进一步的实现经验, 有关名字空间概念的讨论以及另一些应用更增强了我对这个结论的信心。

17.4 一个解决方案：名字空间

所采纳的解决方案从本质上说就是很简单的，它提供了四个新机制：

- 1) 一种定义作用域的机制，即名字空间，用于放置传统上C和C++全局声明的东西。这种作用域可以命名，访问名字空间的成员采用访问类成员的传统记法：`namespace _name:::member _name`。事实上，类作用域可以看成名字空间作用域的特殊情况。
- 2) 一种为名字空间名定义局部同义词的机制。
- 3) 一种允许不显式地写出`namespace _name::`量化词而访问名字空间成员的机制：使用声明。
- 4) 一种允许不显式地写出`namespace _name::`量化词而直接访问名字空间的全部成员的机制：使用指示。

这满足了17.3节的评价准则。进一步说，它还解决了一个长期遗留的问题：从一个派生类的作用域里访问基类成员（见17.5.1节和17.5.2节）；并使对全局名字使用的`static`变成了多余的东西（17.5.3节）。

考虑：

```
namespace A {
    void f(int);
    void f(char);
    class String { /* ... */ };
    // ...
}
```

在名字空间括号里声明的名字就是在名字空间A内部的东西，它们不会与全局名字或者其他任何名字空间里面的名字冲突。名字空间声明（包括定义）与全局声明具有完全相同的语义，只是它们名字的作用域被限制在名字空间内部。

程序员可以通过显式量化的形式，直接使用这些名字：

```
A::String s1 = "Annemarie";

void g1()
{
    A::f(1);
}
```

换一种方式，我们也可以显式地让某个特定库里的个别名字可以不需要量化描述而直接使用，这通过使用声明完成：

```
using A::String;
String s2 = "Nicholas"; // meaning A::String
void g2()
{
    using A::f; // introduce local synonym for A's f
    f(2);      // meaning A::f
}
```

一个使用声明在一个局部作用域为它所提出的名字引进了一个同义词。

再换一种方式，我们也可以显式地要求一个特定库里所有的名字都可以直接使用，不需要借助量化，这通过一个使用指示完成：

```

using namespace A;      // make all names from A accessible
String s3 = "Marian"; // meaning A::String

void g3()
{
    f(3);           // meaning A::f
}

```

使用指示并不为局部作用域引进任何新名字，而只是简单地使有关名字空间里的所有名字都成为可以访问的。

在我原来的设计里，对于使用指示用的是一种更简单更少废话的语法形式：

```
using A; // meaning ``using namespace A;''
```

这使得在使用声明和使用指示之间出现了一种极大的混乱。在我引进更鲜明的语法之后，大部分混乱都消失了。这种更鲜明的语法也使语法分析更简单。

我预计人们需要避免重复写长的名字空间名。因此，在原来的设计里，允许在一个使用声明里列出几个成员的名字：

```
using X::(f,g,h);
```

这在语法上很难看，后来我们考虑过的许多其他形式也与此类似。说得更准确些，我们考虑的每种不同形式都被一些人认为丑得无法忍受。经过名字空间的试用，我发现对这种列表的需要远比我原来预料的少得多。我在读程序时也常常没注意到这种列表，因为它们太像函数声明了。所以我也就转而逐渐建立起使用重复的使用声明的习惯：

```

using X::f;
using X::g;
using X::h;

```

因此，现在没有为描述成员名字的表而用的特殊的使用声明形式。

名字空间为语言特征提供了一个例子，它完全是通过实际经验而被注意到的。名字空间也很容易实现，因为它与C++对作用域和类的观点完全一致。

17.4.1 有关使用名字空间的观点

为访问名字空间里的名字提供了三种方式，这样做一方面是考虑在大程序里的命名问题中多种不可调和的观点，也是经过长时间讨论的结果。一些人强调，要获得可靠的易维护的程序，对每个非局部名字的使用就都应该经过量化。很自然，这些人强调使用显式量化，也表达了他们对使用声明以及更多的是对使用指示的存在价值的疑问。

另一些人则谴责显式量化太啰嗦，实在无法接受。它们使代码的修改太困难，限制了灵活性，并使转变到使用名字空间变得几乎不可能。很自然，这些人争论的目的就是要求有使用指示或者其他机制，以便能将常规的短名字映射到名字空间里。

我同情这两种观点的不那么极端的形式。因此名字空间允许了各种风格，并不强求任何一种东西。局部的风格指南可以——像往常一样——用于强调某些约束，但是以语言规则的方式对全部用户提出要求就很不明智了。

大部分人——相当合理地——担心常规的未量化的名字可能被“劫持”，也就是说，被约束到某个并不是程序员所期望的对象或者函数上。每个C程序员都在这个或那个时间受到过这

种现象的伤害。显式量化可以大大缓解这类问题。另一个类似的，但又很不同的担心是，找到一个名字的声明可能变得非常困难，猜测一个包含它的表达式的意义也可能很困难。显式量化能提供很有力的线索，使人们经常不再去找有关的声明：库的名字加上函数的名字常常能使表达式的意义非常清楚。由于这些原因，在使用不常见的或者使用不频繁的非局部名字时，最好是采用显式量化形式。这样将显著增强代码的清晰性。

在另一方面，显式地量化每个人都知道（或者说是应该知道）的和频繁使用的名字也会变成很讨厌的事情。例如，写 `stdio::printf`、`math::sqrt` 和 `iostream::cout`，对任何了解 C++ 的人都不可能有任何帮助。这些外加的可见杂物很容易变成错误的根源。这个论据强烈要求类似使用声明和使用指示的机制。其中使用声明更易辨别也更少危险性。而使用指示：

```
using namespace X;
```

将使不明的一集名字变得可以使用了。特别是这个指示在今天使某一集名字可以使用了，但如果 X 被修改，明天能用的可能就是另一组不同的名字。那些认为这种情况值得担忧的人们更喜欢明确地用使用声明，列出他们想用的 X 里的名字：

```
using X::f;
using X::g;
using X::h;
```

无论如何，取得对于某个名字空间里每个名字的访问权而不点明它们，让可用名字的集合随 X 的定义而改变，又不必修改用户代码，这种能力通常不大会恰好是人们所希望的。

17.4.2 使名字空间投入使用

由于已经有了成百万行依赖于全局名字和现存库的 C++ 代码，因此，我认为关于名字空间最重要的问题就是如何使名字空间能够投入使用？如果没有一条简单的转变途径，使用户和库的提供商能沿着它去引入名字空间，那么无论基于名字空间的代码有多么优雅，从根本上也是无关紧要的。要求大范围重写绝不是一种选择。

考虑最经典的第一个 C 程序：

```
#include <stdio.h>

int main()
{
    printf("Hello, world\n");
}
```

打破这种程序绝不是一个好主意。我也不认为将标准库弄成一种特殊情况是好主意。我认为，最重要的就是保证名字空间机制能足够好地为标准库服务。在这种方式中，标准化委员会不能为他们自己的库要求任何特权，如果不想把同样的东西扩展开提供给其他库的话。换句话说，如果你也不想生活在某些规则之下，就不要将它们强加给别人。

使用指示是达到这种目标的非常关键的东西。例如，`stdio.h` 将被包裹在下面这样的名字空间里：

```
// stdio.h:

namespace std {
```

```

    // ...
    int printf(const char* ... );
    // ...
}
using namespace std;

```

这就达到了向后的兼容性。另外还定义了一个新的头文件stdio，提供给那些不希望这些名字能够隐含地可用的人们：

```

// stdio:

namespace std {
    // ...
    int printf(const char* ... );
    // ...
}

```

担心重复定义的人们当然可以通过包含stdio的方式定义stdio.h：

```

// stdio.h:

#include<stdio>
using namespace std;

```

按照个人观点，我认为使用指示基本上是一种转变的工具。如果需要引用来自其他名字空间的名字，通过用显式量化和使用声明可以把大部分程序表达得更清晰。

很自然，来自本名字空间的名字不需要量化：

```

namespace A {
    void f();
    void g()
    {
        f(); // call A::f; no qualifier necessary
        // ...
    }
}

void A::f()
{
    g(); // call A::g; no qualifier necessary
    // ...
}

```

在这个方面，名字空间的行为正好和类完全一样。

17.4.3 名字空间的别名

如果用户为他们的名字空间采用了很短的名字，那么不同名字空间的名字也可能发生冲突：

```

namespace A { // short namespace name:
    // will clash (eventually)
    // ...
};

A::String s1 = "asdf";
A::String s2 = "lkjh";

```

但无论如何，长的名字空间名也可能令人生厌：

```
namespace American_Telephone_and_Telegraph { // too long
    // to use in
    // real code
}

// ...

American_Telephone_and_Telegraph::String s3 = "asdf";
American_Telephone_and_Telegraph::String s4 = "lkjh";
```

这种两难问题可以通过为长的名字空间提供短的别名的方式来解决：

```
// use namespace alias to shorten names:

namespace ATT = American_Telephone_and_Telegraph;

ATT::String s3 = "asdf";
ATT::String s4 = "lkjh";
```

这个特征还允许用户引用“某个库”而不必准确说出每次实际上使用的是哪个库。事实上，名字空间也能用于从多个名字空间组合出包含某些名字的界面：

```
namespace My_interface {
    using namespace American_Telephone_and_Telegraph;
    using My_own::String;
    using namespace OI;
    // resolve clash of definitions of 'Flags'
    // from OI and American_Telephone_and_Telegraph:
    typedef int Flags;
    // ...
}
```

17.4.4 利用名字空间管理版本问题

作为名字空间的一个实例，我将显示一个库提供商可以如何利用名字空间去管理版本间互不兼容的变化。这个技术最先是由Tanj Bennett给我指出来的。下面是我Release1：

```
namespace release1 {
    // ...
    class X {
        Impl::Xrep* p;
    public:
        virtual void f1() = 0;
        virtual void f2() = 0;
        // ...
    };
    // ...
}
```

Impl是某个名字空间，我在那里放实现的细节。

用户可能以如下方式写代码：

```
class XX : public release1::X {
    int xx1;
    // ...
public:
```

```

void f1();
void f2();
virtual void ff1();
virtual void ff2();
// ...
);

```

这就意味着我，作为库的提供商，不能改变Release1::X对象的大小（例如增加数据成员），增加或者重新安排虚函数等等，因为这些将隐含地要求用户必须重新做编译，以便能针对我的修改重新调整对象的布局。也存在一些C++的实现能将用户与这种修改隔离开，但是这样的实现并不常见，所以如果作为库提供商的我要依靠这种编译系统，实际上就是把自己与某个编译系统提供商绑到一起了。我可能会用这种方式促使用户不脱离我的库类，但是他们最终还是会那样做。此外，即使给他们提出警告，他们也会对重新编译提出疑义。

我需要一种更好的解决方案。利用名字空间来区分不同的版本，我的Release2可能具有下面这种样子：

```

namespace release1 { // release1 supplied for compatibility
// ...
class X {
    Impl::Xrep* p; // Impl::Xrep has changed
                    // to accommodate release2
public:
    virtual void f1() = 0;
    virtual void f2() = 0;
    // ...
};
// ...
}

namespace release2 {
// ...
class X {
    Impl::Xrep* p;
public:
    virtual void f2() = 0; // new ordering
    virtual void f3() = 0; // more functions
    virtual void f1() = 0;
    // ...
};
// ...
}

```

老代码使用Release1，而新代码使用Release2。新老代码不仅都可以工作，而且还可以共存。Release1和Release2的头文件也分开，使用户只需要#include最少的东西。为使升级的工作更简单，用户可以通过名字空间别名将版本变化的影响局部化。一个简单文件：

```

// lib.h:
namespace lib = release1;
// ...

```

能以与版本无关的方式包含所有的东西，以下面的形式用在任何地方：

```
#include "lib.h"
```

```
class XX : public lib::X {
    // ...
};
```

通过一个简单的修改就可以升级到新版本了：

```
// lib.h:
namespace lib = release2;
// ...
```

只是到了某个时候，由于需要使用Release2，或是要做重新编译，或是需要去处理某些版本间的源代码不兼容问题，这时才需要去做升级。

17.4.5 细节

本小节将讨论与作用域解析、全局作用域、重载、嵌套的名字空间以及用一些分离的部分组合名字空间等有关的一些技术细节。

1. 方便性与安全性

使用声明是向局部作用域里添加东西。而使用指示并不添加任何东西，它只是使一些名字能够被访问，例如：

```
namespace X {
    int i, j, k;
}

int k;

void f1()
{
    int i = 0;
    using namespace X; // make names from X accessible
    i++;             // local i
    j++;             // X::j
    k++;             // error: X::k or global k ?
    ::k++;           // the global k
    X::k++;          // X's k
}
void f2()
{
    int i = 0;
    using X::i; // error: i declared twice in f2()
    using X::j;
    using X::k; // hides global k

    i++;
    j++;           // X::j
    k++;           // X::k
}
```

这样也就维持了一种非常重要的性质：局部声明的名字（无论是通过正常的局部声明所声明的，还是通过使用声明）遮蔽名字相同的非局部声明，而名字的任何非法重载都将在声明点被检查出来。

如上所示，在全局作用域中，在可访问方面并不给全局作用域任何超越名字空间的优先权，这就为防止偶然的名字冲突提供了某种保护。

在另一方面，非局部的名字在它们的声明所在的上下文里寻找和处理，就像其他非局部名字一样，特别是与使用指示有关的错误都只在使用点检查。这样就能帮助程序员，不因潜在的错误而出现程序失败。例如：

```
namespace A {
    int x;
}

namespace B {
    int x;
}

void f()
{
    using namespace A;
    using namespace B; // ok: no error here

    A::x++; // ok
    B::x++; // ok
    x++; // error: A::x or B::x ?
}
```

2. 全局作用域

引进名字空间之后，全局作用域就变成了另一个名字空间。全局名字空间的独特之处仅在于你不必在显式量化形式中写出它的名字。`::f`的意思是“在全局作用域里声明的那个`f`”，而`X::f`意味着“在名字空间`X`里声明的那个`f`”。考虑：

```
int a;

void f()
{
    int a = 0;
    a++; // local a
    ::a++; // global a
}
```

如果我们将它包裹到一个名字空间里，再加上另一个名字为`a`的变量，得到的是：

```
int a;

namespace X {
    int a;

    void f()
    {
        int a = 0;
        a++; // local a
        X::a++; // X::a
        ::a++; // X::a or global a ? -- the global a !
    }
}
```

也就是说，用`::`量化意味着“全局”，而不是“外面包裹的最近名字空间”。如果采用后一种规定，将能保证将任意代码包裹到一个名字空间里时不会改变代码的意义。但是，那样做全局名字空间就不会有名字了，这将无法与下面观点保持一致：全局名字空间也是一个名字空

间，只不过是一个奇特的名字。因此我们还是采用了前一个意思，所以`::a`引用的是全局作用域里声明的`a`。

我期望能够看到全局名字的使用急剧减少。有关名字空间的规则都特别做了加工，以保证，与那些细心地防止污染全局空间的人们相比，随意使用全局名字的“懒散”用户将不可能得到更多的利益。

请注意，使用指示并不在它所出现的作用域里声明任何名字：

```
namespace X {
    int a;
    int b;
    // ...
}

using namespace X; // make all names from X accessible
using X::b;        // declare local synonym for X::b

int i1 = ::a; // error: no ``a'' declared in global scope
int i2 = ::b; // ok: find the local synonym for X::b
```

这就意味着，如果把一个全局的库放进名字空间里，就会打破那些显式地用`::`去访问库函数的老代码。解决的方法或者是修改代码，显式地提出新库名字空间的名字；或者是引进适当的全局使用声明。

3. 重载

在有关名字空间的建议中，最有争议的部分就是决定依据常规的重载规则，允许跨名字空间的重载。考虑：

```
namespace A {
    void f(int);
    // ...
}
using namespace A;

namespace B {
    void f(char);
    // ...
}
using namespace B;

void g()
{
    f('a'); // calls B::f(char)
}
```

如果一个用户没有仔细查看名字空间`B`，就可能期望被调用的是`A::f(int)`。更糟的是，一个去年曾经仔细查看过这个程序的用户，如果没有注意到在后一个版本里的`B`中增加了一个`f(char)`声明，也可能会大吃一惊。

但无论如何，这种问题只出现在你维护一个程序，其中在同一个作用域里显式地两次使用`using namespace`——对于新写的软件，这是一种不应推荐的方式。一个函数调用在不同名字空间里存在的两个合法解析，这也是一个优化的编译程序明显应该提出警告的地方，即使按照常规的重载规则其中的某个解析比另一个更好些。我基本上把使用指示看作是一种

转变的辅助手段，理论上说，写新代码的人可以避免大部分这类东西，剩下的若干实际问题可以尽量通过显式量化或使用声明去处理。

对于允许跨名字空间重载的问题，我的理由是存在着最简单的规则（“使用常规的重载规则”），而且它也是我所能想到的，使我们能从现存的库迁移到使用名字空间，而且只需要对代码做最少修改的惟一规则。例如：

```
// old code:

void f(int);    // from A.h
// ...

void f(char);  // from B.h
// ...

void g()
{
    f('a'); // calls the f from B.h
}
```

很容易升级到上面所示的使用了名字空间的版本，除了头文件之外什么都不需要修改。

4. 嵌套的名字空间

名字空间的一个明显用途就是将完整的一集声明和定义包裹到一个名字空间里：

```
namespace X {
    // all my declarations
}
```

一般地说，在这些声明里也可以包括名字空间。这样，为了某些实际的原因——就像由于最简单的原因，应该允许结构的嵌套，除非有特别强的理由说明不应该这样做——应该允许嵌套的名字空间。例如：

```
void h();

namespace X {
    void g();
    // ...
    namespace Y {
        void f();
        void ff();
        // ...
    }
    // ...
}
```

常规的作用域和量化规则照常适用：

```
void X::Y::ff()
{
    f(); g(); h();
}

void X::g()
{
    f(); // error: no f() in X
```

```

    Y::f();
}

void h()
{
    f();           // error: no global f()
    Y::f();        // error: no global Y
    X::f();        // error: no f() in X
    X::Y::f();
}

```

5. 名字空间是开放的

名字空间是开放的，也就是说，你可以在多个名字空间声明中将名字加进一个名字空间里。例如：

```

namespace A {
    int f(); // now A has member f()
};

namespace A {
    int g(); // now A has two members f() and g()
}

```

这样做的目的就是为了支持在一个名字空间里有一些大的程序片段，其方式与当前许多库或应用都生存在同一个全局名字空间里一样。为了达到这个目的，就应该允许名字空间的定义散布在多个头文件和源程序文件里。这种开放性也被看作是一种转变的辅助手段。例如：

```

// my header:
extern void f(); // my function
// ...
#include<stdio.h>
extern int g(); // my function
// ...

```

可以重新写成下面的形式，不必调换声明的顺序：

```

// my header:

namespace Mine {
    void f(); // my function
    // ...
}

#include<stdio.h>

namespace Mine {
    int g(); // my function
    // ...
}

```

当前流行的口味（包括我自己）是喜爱用许多小的名字空间，而不是将大量的代码片段放进一个名字空间里。可以通过要求将所有成员都在一个名字空间声明里的方式去强制要求这种风格，就像所有类成员都必须写在一个类声明里一样。但是，我看不到提供这么多小规矩的理由。与迎合某些当前口味的更限制的系统相比，我还是更喜欢开放的名字空间。

17.5 对于类的影响

也有人建议将名字空间作为一种特殊的类。我不认为这是个好主意，因为类的许多功能就是为了将类作为一种用户定义类型的概念而存在的。例如，为创建和操作这种类型的对象所定义的那些功能与作用域问题根本就没有关系。

反过来说，把类看作是一种名字空间看起来更像是对的。一个类在某种意义上就是一个名字空间，对名字空间可以做的所有操作都能够在同样意义下应用到类上，除非某个操作是类所明确禁止的。这也意味着简单性和普遍性，而且能使实现工作减到最小。我认为，这种观点已经由名字空间机制的平滑引入而得到了证明。但也很明显，这种解决方案与随着基本名字空间机制而产生的长远问题无关。

17.5.1 派生类

考虑一个老问题，一个类的成员将遮蔽其基类里具有同样名字的成员：

```
class B {
public:
    f(char);
};

class D : public B {
public:
    f(int); // hides f(char)
};

void f(D& d)
{
    d.f('c'); // calls D::f(int)
}
```

自然，引进名字空间并不会改变这个例子的意义，但是却可能为它做出一种新解释：由于D是一个类，它所提供的作用域是个名字空间。而名字空间D是嵌套在名字空间B的里面，所以D::f(int)将遮蔽B::f(char)，所以被调用的是D::f(int)。如果这个解析不是我们所希望的，那么就可以通过一个使用声明，将B的f()引入到作用域里：

```
class B {
public:
    f(char);
};

class D : public B {
public:
    f(int);
    using B::f; // bring B::f into D to enable overloading
};

void f(D& d)
{
    d.f('c'); // calls D::f(char) !
}
```

我们突然有了一种选择^Θ。

与以往一样，来自不同兄弟类的名字可能造成歧义性（与它们的名字无关）：

```
struct A { void f(int); };
struct B { void f(double); };

struct C : A, B {
    void g() {
        f(1);      // error: A::f(int) or B::f(double)
        f(1.0);    // error: A::f(int) or B::f(double)
    }
};
```

但是如果我们想消解这种歧义性，现在就能够做了：只要加一对使用声明，将A::f和B::f都引入到作用域C中：

```
struct C : A, B {
    using A::f;
    using B::f;

    void g() {
        f(1);      // A::f(1)
        f(1.0);    // B::f(1.0)
    }
};
```

过去几年中反复出现过有关这个方向上的显式机制的建议。我还记得在Release 2.0工作期间，与Jonathan Shapipo讨论了关于这方面的可能性，但最后因为将它包括进来“太特殊且惟一”而将之拒绝了。而在另一方面，使用声明是一种一般性的机制，正好又为这个问题提供了一种解决的方法。

17.5.2 使用基类

为了避免混乱，作为类成员的使用声明必须用名字（直接或者间接地）提出基类的一个成员。为了避免与支配规则（12.3.1节）发生问题，不允许将使用指示作为类的成员。

```
struct D : public A {
    using namespace A; // error: using-directive as member
    using ::f;        // error: ::f not a member of a base class
};
```

直接指出基类成员名字的使用声明在调整访问方面能扮演一种很重要的角色：

```
class B {
public:
    f(char);
};

class D : private B {
public:
    using B::f;
};
```

^Θ 上面程序段最后函数注释里的D::f(char)应该是B::f(char)，原书有误。——译者注

这就得到了一种更一般也更清晰的方式，完成了引入访问声明（2.10节）所要做的事情：

```
class D : private B {
public:
    B::f; // old way: access declaration
};
```

这样，使用声明实际上已经使访问声明成为多余的东西。因此访问声明将受到抑制，也就是说，将访问声明贬低为在不久的将来，在用户有充沛时间升级之后就要删除的东西。

17.5.3 清除全局的static

将一集声明包裹到一个名字空间里，简单地说就是为了避免与头文件里的声明互相干扰，或者为了避免自己使用的名字与其他编译单元里的全局声明相互干扰，这些也是很有意义的。例如：

```
#include <header.h>
namespace Mine {
    int a;
    void f() { /* ... */ }
    int g() { /* ... */ }
}
```

在许多情况下，我们对于名字空间本身采用什么名字并不在意，只要它不与其他名字空间的名字相互冲突。为了更漂亮地服务于这种用途，我们也允许匿名的名字空间：

```
#include <header.h>
namespace {
    int a;
    void f() { /* ... */ }
    int g() { /* ... */ }
}
```

除了不会与头文件里的名字产生重载之外，这等价于：

```
#include <header.h>

static int a;
static void f() { /* ... */ }
static int g() { /* ... */ }
```

而这种重载通常都不是人们所希望的。如果真需要的话也很容易做到：

```
namespace {
#include <header.h>
    int a;
    void f() { /* ... */ }
    int g() { /* ... */ }
}
```

这样，名字空间的概念就使我们可以抑制static在控制全局名字的可见性方面的使用。这就会使static在C++里只剩下一个意义：静态地分配，且不重复。

这种匿名名字空间在其他方面与别的名字空间完全一样，只是我们不需要去说它的名字罢了。简单地说：

```
namespace { /* ... */ }
```

等价于：

```
namespace unique_name { /* ... */ }
using namespace unique_name;
```

在一个作用域里的各个匿名名字空间都共享同样的惟一名字。特别是在一个编译单元中所有全局的匿名名字空间都是同一个名字空间的一部分，而它们又与其他编译单元的匿名名字空间不同。

17.6 与C语言的兼容性

具有C连接的函数也可以放进一个名字空间里：

```
namespace X {
    extern "C" void f(int);
    void g(int)
}
```

这就使具有C连接的函数可以像名字空间的其他成员一样使用。例如：

```
void h()
{
    X::f();
    X::g();
}
```

当然，在同一个程序里，不能在两个不同的名字空间里存在两个同样名字的具有C连接的函数，因为它们将被解析到同一个C函数。C语言不安全的连接规则将使这种错误很难发现。

换一种方式设计，就是不允许在名字空间里出现具有C连接的函数。这将导致人们不使用名字空间，因为这实际上迫使需要与C函数接口的人们去污染全局的名字空间。这种根本不是解决办法的东西，当然是无法接受的。

另一种选择，就是保证在两个不同名字空间里的同名函数总是不同的实际函数，即使它们具有C连接。例如：

```
namespace X {
    extern "C" void f(int);
}

namespace Y {
    extern "C" void f(int);
}
```

问题是如何从一个C程序里调用这种函数。因为C语言里没有基于名字空间的歧义性消解机制，我们将只能依赖于某种（肯定是与实现有关的）命名规则。例如，这个C程序可能不得不去访问__X__f和__Y__f。这种解决方法也被认为是不可接受的，我们被C语言的不安全的规则卡住了。C将污染连接程序的名字空间，但不会影响C++编译单位的全局名字空间。

请注意，这是C语言的一个问题（一个兼容性的窘境），而不是C++名字空间的问题。连接到一个具有类似于C++名字空间机制的语言，一切都是显然的和安全的。例如，我期望下面的东西。

```
namespace X {
    extern "Ada" void f(int);
}

namespace Y {
    extern "Ada" void f(int);
}
```

是从C++程序映射到不同Ada程序包里有关函数的方式。

第18章 C语言预处理器

而且，我还持有这样的观点，

Cpp必须被摧毁。

——老加图(*Marcus Porcius Cato^②*)

C语言预处理器(Cpp)的问题——Cpp结构的替代品——禁止Cpp

Cpp

在C++从C语言那里继承来的功能、技术和思想中也包括C的预处理器，Cpp。我原来就不喜欢Cpp，现在也不喜欢它。这个预处理器具有字符和文件的本性，这些从根本上说与一个围绕着作用域、类型和界面等概念设计出来的程序设计语言是格格不入的。例如，考虑下面看起来完全无害的代码片段：

```
#include<stdio.h>
extern double sqrt(double);

main()
{
    printf("The square root of 2 is %g\n",sqrt(2));
    fflush(stdout);
    return(0);
}
```

它会做些什么？打印出

```
The square root of 2 is 1.41421
```

可能吗？看起来是可能的，但如果我实际上这样编译它：

```
cc -Dsqrt=rand -Dreturn=abort
```

它就会打印出

```
The square root of 2 is 7.82997e+28
abort - core dumped
```

并将一个内核映像留了下来^②。

这个例子可能有些极端，你也可能认为，用编译选项去定义Cpp的宏不那么光明正大，但是这种例子并不是不实际的。宏定义可能潜伏在环境、编译指示和头文件里，宏替换可以穿透所有的作用域边界，确实可以改变程序作用域的结构（通过插入花括号或引号等），也允许

② 老加图，公元前234—149，古罗马政治家。*Cato*的一句名言是“*Delenda est Carthage*”，意为“迦太基必须被摧毁”，迦太基是当时北非的一个奴隶制国家，在今天突尼斯境内。老加图自然不关心Cpp的任何事情。作者在这里是模仿老加图的这句名言。——译者注

③ 在UNIX等系统里，程序的非正常终止将导致系统自动把当时的内存现场（成为内存映像）保存为一个文件，供人检查。C标准库函数abort将导致程序非正常终止。——译者注

程序员根本不接触源代码的情况下改变编译程序真正看到的东西。即使是Cpp最极端的使用，有时可能也是有用的。但是，它的功能是如此的非结构化、如此生涩强硬，这就使它们将永远是程序员、维护者、移植代码和建造工具的人们头疼的问题。

回过头看，Cpp最坏的方面可能就是它击垮了C语言程序设计环境的开发。Cpp无政府主义的字符层次的操作，使任何不太简单的C和C++工具都比人们可能设想的更大、更慢、更不优雅，也更低效。

Cpp甚至也不是一个好的宏处理器。因此我早就定下目标，要使Cpp成为多余的东西。后来发现这个工作比自己所期望的要困难得多。Cpp可能是极其丑陋的，但却很难为它的丰富应用方式找到具有更好结构而又高效的替代品。

C预处理器最基本的指示字有四个^Θ：

- 1) #include从其他文件里复制源程序正文。
- 2) #define定义宏（有参数的或没有参数的）。
- 3) #ifdef根据某个条件确定是否应该包括一些代码行。
- 4) #pragma以某种与实现有关的方式影响编译过程。

这些指示字被用于表述各种各样的基本程序设计工作：

```
#include
    — 使界面定义可以使用
    — 组合源程序正文

#define
    — 定义符号常量
    — 定义宏子程序
    — 定义类属子程序
    — 定义类属“类型”
    — 重新命名
    — 字符串拼接
    — 定义专用的语法
    — 一般性的宏处理

#ifndef
    — 版本控制
    — 注释掉一些代码

#pragma
    — 布局控制
    — 为编译程序提供非常规的控制流信息
```

Cpp对于所有这些工作做得都很不好，常常是通过某种间接的方式，但是却很廉价，常常也恰如其分。最重要的是，在任何有C的地方Cpp都能用，而且人人皆知。这就使它比那些虽然更好，但却不那么广泛可用、广泛知晓的宏处理器更有用。这个方面是如此重要，以至于C预处理器常常被用于一些与C语言没什么关系的事项。但这不是C++的问题。

C++为#define的主要应用提供了如下的替代方式：

^Θ #if、#line和#undef指示字可能也很重要，但是并不影响这里的讨论。

- `const` 用于常量 (3.8节)。
- `inline` 用于开子程序 (2.4.1节)。
- `template` 用于以类型为参数的函数 (15.6节)。
- `template` 用于参数化类型 (15.3节)。
- `namespace` 用于更一般的命名问题 (第17章)。

C++ 没有为`#include`提供替代形式，虽然名字空间提供了一种作用域机制，它能以某种方式支持组合，利用它可以改善`#include`的行为方式。

我曾经建议可以给C++本身增加一个`include`指示字，作为Cpp的`#include`的替代品。C++的这种`include`可以在下面三个方面与Cpp的`#include`不同：

- 1) 如果一个文件被`include`两次，第二个`include`将被忽略。这解决了一个实际问题，而目前这个问题是通过`#define`和`#ifdef`，以非常低效而笨拙的方式处理的。
- 2) 在`include`的正文之外定义的宏将不在`include`的正文内部展开。这就提供了一种能够隔离信息的机制，可以使这些信息不受宏的干扰。
- 3) 在`include`的正文内部定义的宏在`include`正文之后的正文处理中不展开。这保证了`include`正文内部的宏不会给包含它的编译单位强加上某种顺序依赖性，并一般地防止了由宏引起的奇怪情况。

对于采用预编译头文件的系统而言，一般地说，对于那些要用独立部分组合软件的人们而言，这种机制都将是一个福音。请注意，无论如何这还只是一个思想而不是一个语言特征。

留下的是`#ifdef`和`#progma`。没有`#progma`我也能活，因为我还没有见过一个自己喜欢的`#progma`。看起来`#progma`被过分经常地用于将语言语义的变形隐藏到编译系统里，或者被用于提供带有特殊语义和笨拙语法的语言扩充。我们至今还没有`#ifdef`的很好的替代品。特别是用`if`语句和常量表达式还不足以替代它。例如：

```
const C = 1;

// ...

if (C) {
    // ...
}
```

这种技术不能用于控制声明，而且一个`if`语句的正文必须在语法上正确，满足类型检查，即使它处在一个绝不会被执行的分支里。

我很愿意看到Cpp被废除。但无论如何，要想做到这件事，惟一现实的和负责任的方式就是首先使它成为多余的，而后鼓励人们去使用那些更好的替代品，最后——在许多年之后——将Cpp放逐到程序开发环境里，与其他附加性语言工具放到一起，那里才是它应该待的地方。

参 考 文 献

- [2nd] see [Stroustrup,1991].
- [Agha,1986] Gul Agha: *An Overview of Actor languages*. ACM SIGPLAN Notices. October 1986.
- [Aho,1986] Alfred Aho, Ravi Sethi, and Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA. 1986. ISBN 0-201-10088-6.
- [ARM] see [Ellis,1990].
- [Babciský,1984] Karel Babciský: *Simula Performance Assessment*. Proc. IFIP WG2.4 Conference on System Implementation Languages: Experience and Assessment. Canterbury, Kent, UK. September 1984.
- [Barton,1994] John J. Barton and Lee R. Nackman: *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison-Wesley, Reading, MA. 1994. ISBN 0-201-53393-6.
- [Birtwistle,1979] Graham Birtwistle, Ole-Johan Dahl, Björn Myrhaug, and Kristen Nygaard: *SIMULA BEGIN*. Studentlitteratur, Lund, Sweden. 1979. ISBN 91-44-06212-5.
- [Boehm,1993] Hans-J. Boehm: *Space Efficient Conservative Garbage Collection*. Proc. ACM SIGPLAN '93 Conference on Programming Language Design and Implementation. ACM SIGPLAN Notices. June 1993.
- [Booch,1990] Grady Booch and Michael M. Vilot: *The Design of the C++ Booch Components*. Proc. OOPSLA'90. October 1990.
- [Booch,1991] Grady Booch: *Object-Oriented Design*. Benjamin Cummings, Redwood City, CA. 1991. ISBN 0-8053-0091-0.
- [Booch,1993] Grady Booch: *Object-oriented Analysis and Design with Applications, 2nd edition*. Benjamin Cummings, Redwood City, CA. 1993. ISBN 0-8053-5340-2.
- [Booch,1993b] Grady Booch and Michael M. Vilot: *Simplifying the C++ Booch Components*. The C++ Report. June 1993.
- [Budge,1992] Ken Budge, J.S. Perry, and A.C. Robinson: *High-Performance Scientific Computation using C++*. Proc. USENIX C++ Conference. Portland, OR. August 1992.
- [Buhr,1992] Peter A. Buhr and Glen Ditchfield: *Adding Concurrency to a Programming Language*. Proc. USENIX C++ Conference. Portland, OR. August 1992.
- [Call,1987] Lisa A. Call, et al.: *CLAM - An Open System for Graphical User Interfaces*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
- [Cameron,1992] Don Cameron, et al.: *A Portable Implementation of C++ Exception Handling*. Proc. USENIX C++ Conference. Portland, OR. August 1992.

- [Campbell,1987] Roy Campbell, et al.: *The Design of a Multiprocessor Operating System*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
- [Cattell,1991] Rich G.G. Cattell: *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Addison-Wesley, Reading, MA. 1991. ISBN 0-201-53092-9.
- [Cargill,1991] Tom A. Cargill: *The Case Against Multiple Inheritance in C++*. USENIX Computer Systems. Vol 4, no 1, 1991.
- [Carroll,1991] Martin Carroll: *Using Multiple Inheritance to Implement Abstract Data Types*. The C++ Report. April 1991.
- [Carroll,1993] Martin Carroll: *Design of the USL Standard Components*. The C++ Report. June 1993.
- [Chandy,1993] K. Mani Chandy and Carl Kesselman: *Compositional C++: Compositional Parallel Programming*. Proc. Fourth Workshop on Parallel Computing and Compilers. Springer-Verlag. 1993.
- [Cristian,1989] Flaviu Cristian: *Exception Handling*. Dependability of Resilient Computers, T. Andersen, editor. BSP Professional Books, Blackwell Scientific Publications, 1989.
- [Cox,1986] Brad Cox: *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, Reading, MA. 1986.
- [Dahl,1988] Ole-Johan Dahl: Personal communication.
- [Dearle,1990] Fergal Dearle: *Designing Portable Applications Frameworks for C++*. Proc. USENIX C++ Conference. San Francisco, CA. April 1990.
- [Dorward,1990] Sean M. Dorward, et al.: *Adding New Code to a Running Program*. Proc. USENIX C++ Conference. San Francisco, CA. April 1990.
- [Eick,1991] Stephen G. Eick: *SIMLIB - An Object-Oriented C++ Library for Interactive Simulation of Circuit-Switched Networks*. Proc. Simulation Technology Conference. Orlando, FL. October 1991.
- [Ellis,1990] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA. 1990. ISBN 0-201-51459-1.
- [Faust,1990] John E. Faust and Henry M. Levy: *The Performance of an Object-Oriented Threads Package*. Proc. ACM joint ECOOP and OOPSLA Conference. Ottawa, Canada. October 1990.
- [Fontana,1991] Mary Fontana and Martin Neath: *Checked Out and Long Overdue: Experiences in the Design of a C++ Class Library*. Proc. USENIX C++ Conference. Washington, DC. April 1991.
- [Forslund,1990] David W. Forslund, et al.: *Experiences in Writing Distributed Particle Simulation Code in C++*. Proc. USENIX C++ Conference. San Francisco, CA. April 1990.
- [Gautron,1992] Philippe Gautron: *An Assertion Mechanism based on Exceptions*. Proc. USENIX C++ Conference. Portland, OR. August 1992.
- [Gehani,1988] Narain H. Gehani and William D. Roome: *Concurrent C++: Concurrent Programming With Class(es)*. Software—Practice & Experience. Vol 18, no 12, 1988.
- [Goldberg,1983] Adele Goldberg and David Robson: *Smalltalk-80, The Lan-*

- [Goodenough,1975] John Goodenough: *Exception Handling: Issues and a Proposed Notation*. Communications of the ACM. December 1975.
- [Gorlen,1987] Keith E. Gorlen: *An Object-Oriented Class Library for C++ Programs*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
- [Gorlen,1990] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico: *Data Abstraction and Object-Oriented Programming in C++*. Wiley. West Sussex. England. 1990. ISBN 0-471-92346-X.
- [Hübel,1992] Peter Hübel and J.T. Thorsen: *An Implementation of a Persistent Store for C++*. Computer Science Department. Aarhus University, Denmark. December 1992.
- [Ichbiah,1979] Jean D. Ichbiah, et al.: *Rationale for the Design of the ADA Programming Language*. SIGPLAN Notices Vol 14, no 6, June 1979 Part B.
- [Ingalls,1986] Daniel H.H. Ingalls: *A Simple Technique for Handling Multiple Polymorphism*. Proc. ACM OOPSLA Conference. Portland, OR. November 1986.
- [Interrante,1990] John A. Interrante and Mark A. Linton: *Runtime Access to Type Information*. Proc. USENIX C++ Conference. San Francisco 1990.
- [Johnson,1992] Steve C. Johnson: Personal communication.
- [Johnson,1989] Ralph E. Johnson: *The Importance of Being Abstract*. The C++ Report. March 1989.
- [Keffer,1992] Thomas Keffer: *Why C++ Will Replace Fortran*. C++ Supplement to Dr. Dobbs Journal. December 1992.
- [Keffer,1993] Thomas Keffer: *The Design and Architecture of Tools.h++*. The C++ Report. June 1993.
- [Kernighan,1976] Brian Kernighan and P.J. Plauger: *Software Tools*. Addison-Wesley, Reading, MA. 1976. ISBN 0-201-03669.
- [Kernighan,1978] Brian Kernighan and Dennis Ritchie: *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ. 1978. ISBN 0-13-110163-3.
- [Kernighan,1981] Brian Kernighan: *Why Pascal is not my Favorite Programming Language*. AT&T Bell Labs Computer Science Technical Report No 100. July 1981.
- [Kernighan,1984] Brian Kernighan and Rob Pike: *The UNIX Programming Environment*. Prentice-Hall, Englewood Cliffs, NJ. 1984. ISBN 0-13-937699-2.
- [Kernighan,1988] Brian Kernighan and Dennis Ritchie: *The C Programming Language (second edition)*. Prentice-Hall, Englewood Cliffs, NJ. 1988. ISBN 0-13-110362-8.
- [Kiczales,1992] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow: *The Art of the Metaobject Protocol*. The MIT Press. Cambridge, Massachusetts. 1991. ISBN 0-262-11158-6.
- [Koenig,1988] Andrew Koenig: *Associative arrays in C++*. Proc. USENIX Conference. San Francisco, CA. June 1988.
- [Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible – but no closer*. The C++ Report. July 1989.

- [Koenig,1989b] Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++*. Proc. "C++ at Work" Conference. November 1989.
- [Koenig,1990] Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++ (revised)*. Proc. USENIX C++ Conference. San Francisco, CA. April 1990. Also, *Journal of Object-Oriented Programming*. July 1990.
- [Koenig,1991] Andrew Koenig: *Applicators, Manipulators, and Function Objects*. C++ Journal, vol. 1, #1. Summer 1990.
- [Koenig,1992] Andrew Koenig: *Space Efficient Trees in C++*. Proc. USENIX C++ Conference. Portland, OR. August 1992.
- [Krogdahl,1984] Stein Krogdahl: *An Efficient Implementation of Simula Classes with Multiple Prefixing*. Research Report No 83. June 1984. University of Oslo, Institute of Informatics.
- [Lea,1990] Doug Lea and Marshall P. Cline: *The Behavior of C++ Classes*. Proc. ACM SOOPPA Conference. September 1990.
- [Lea,1991] Doug Lea: Personal Communication.
- [Lea,1993] Doug Lea: *The GNU C++ Library*. The C++ Report. June 1993.
- [Lenkov,1989] Dmitry Lenkov: *C++ Standardization Proposal*. #X3J11/89-016.
- [Lenkov,1991] Dmitry Lenkov, Michey Mehta, and Shankar Unni: *Type Identification in C++*. Proc. USENIX C++ Conference. Washington, DC. April 1991.
- [Linton,1987] Mark A. Linton and Paul R. Calder: *The Design and Implementation of InterViews*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
- [Lippman,1988] Stan Lippman and Bjarne Stroustrup: *Pointers to Class Members in C++*. Proc. USENIX C++ Conference. Denver, CO. October 1988.
- [Liskov,1979] Barbara Liskov, et al.: *CLU Reference manual*. MIT/LCS/TR-225. October 1979.
- [Liskov,1987] Barbara Liskov: *Data Abstraction and Hierarchy*. Addendum to Proceedings of OOPSLA'87. October 1987.
- [Madsen,1993] Ole Lehrmann Madsen, et al.: *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, Reading, MA. 1993. ISBN 0-201-62430.
- [McCluskey,1992] Glen McCluskey: *An Environment for Template Instantiation*. The C++ Report. February 1992.
- [Meyer,1988] Bertrand Meyer: *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ. 1988. ISBN 0-13-629049.
- [Miller,1988] William M. Miller: *Exception Handling without Language Extensions*. Proc. USENIX C++ Conference. Denver CO. October 1988.
- [Mitchell,1979] James G. Mitchell, et.al.: *Mesa Language Manual*. XEROX PARC, Palo Alto, CA. CSL-79-3. April 1979.
- [Murray,1992] Rob Murray: *A Statically Typed Abstract Representation for C++ Programs*. Proc. USENIX C++ Conference. Portland, OR. August 1992.
- [Nelson,1991] Nelson, G. (editor): *Systems Programming with Modula-3*.

- [Rose,1984] Prentice-Hall, Englewood Cliffs, NJ. 1991. ISBN 0-13-590464-1.
- [Parrington,1990] Leonie V. Rose and Bjarne Stroustrup: *Complex Arithmetic in C++*. Internal AT&T Bell Labs Technical Memorandum. January 1984. Reprinted in AT&T C++ Translator Release Notes. November 1985.
- [Parrington,1990] Graham D. Parrington: *Reliable Distributed Programming in C++*. Proc. USENIX C++ Conference. San Francisco, CA. April 1990.
- [Reiser,1992] John F. Reiser: *Static Initializers: Reducing the Value-Added Tax on Programs*. Proc. USENIX C++ Conference. Portland, OR. August 1992.
- [Richards,1980] Martin Richards and Colin Whitby-Strevens: *BCPL – the language and its compiler*. Cambridge University Press, Cambridge, England. 1980. ISBN 0-521-21965-5.
- [Rovner,1986] Paul Rovner: *Extending Modula-2 to Build Large, Integrated Systems*. IEEE Software Vol 3, No 6, November 1986.
- [Russo,1988] Vincent F. Russo and Simon M. Kaplan: *A C++ Interpreter for Scheme*. Proc. USENIX C++ Conference. Denver, CO. October 1988.
- [Russo,1990] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell: *C++ and Operating Systems Performance: A Case Study*. Proc. USENIX C++ Conference. San Francisco, CA. April 1990.
- [Sakkinen,1992] Markku Sakkinen: *A Critique of the Inheritance Principles of C++*. USENIX Computer Systems, vol 5, no 1, Winter 1992.
- [Sethi,1980] Ravi Sethi: *A case study in specifying the semantics of a programming language*. Seventh Annual ACM Symposium on Principles of Programming Languages. January 1980.
- [Sethi,1981] Ravi Sethi: *Uniform Syntax for Type Expressions and Declarations*. Software – Practice and Experience, Vol 11. 1981.
- [Sethi,1989] Ravi Sethi: *Programming Languages – Concepts and Constructs*. Addison-Wesley, Reading, MA. 1989. ISBN 0-201-10365-6.
- [Shopiro,1985] Jonathan E. Shopiro: *Strings and Lists for C++*. AT&T Bell Labs Internal Technical Memorandum. July 1985.
- [Shopiro,1987] Jonathan E. Shopiro: *Extending the C++ Task System for Real-Time Control*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
- [Shopiro,1989] Jonathan E. Shopiro: *An Example of Multiple Inheritance in C++: A Model of the Iostream Library*. ACM SIGPLAN Notices. December 1989.
- [Schwarz,1989] Jerry Schwarz: *Iostreams Examples*. AT&T C++ Translator Release Notes. June 1989.
- [Snyder,1986] Alan Snyder: *Encapsulation and Inheritance in Object-Oriented Programming Languages*. Proc. OOPSLA'86. September 1986.
- [Stal,1993] Michael Stal and Uwe Steinmüller: *Generic Dynamic Arrays*. The C++ Report. October 1993.
- [Stepanov,1993] Alexander Stepanov and David R. Musser: *Algorithm-Oriented*

- [Stroustrup,1978] *Generic Software Library Development*. HP Laboratories Technical Report HPL-92-65. November 1993.
- [Stroustrup,1979] Bjarne Stroustrup: *On Unifying Module Interfaces*. ACM Operating Systems Review Vol 12 No 1. January 1978.
- [Stroustrup,1979b] Bjarne Stroustrup: *Communication and Control in Distributed Computer Systems*. Ph.D. thesis, Cambridge University, 1979.
- [Stroustrup,1979c] Bjarne Stroustrup: *An Inter-Module Communication System for a Distributed Computer System*. Proc. 1st International Conf. on Distributed Computing Systems. October 1979.
- [Stroustrup,1980] Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. Bell Laboratories Computer Science Technical Report CSTR-84. April 1980. Revised, August 1981. Revised yet again and published as [Stroustrup,1982].
- [Stroustrup,1980b] Bjarne Stroustrup: *A Set of C Classes for Co-routine Style Programming*. Bell Laboratories Computer Science Technical Report CSTR-90. November 1980.
- [Stroustrup,1981]
* Bjarne Stroustrup: *Long Return: A Technique for Improving The Efficiency of Inter-Module Communication*. Software Practice and Experience. January 1981.
- [Stroustrup,1981b] Bjarne Stroustrup: *Extensions of the C Language Type Concept*. Bell Labs Internal Memorandum. January 1981.
- [Stroustrup,1982] Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. ACM SIGPLAN Notices. January 1982. Revised version of [Stroustrup,1980].
- [Stroustrup,1982b] Bjarne Stroustrup: *Adding Classes to C: An Exercise in Language Evolution*. Bell Laboratories Computer Science internal document. April 1982. Software: Practice & Experience, Vol 13. 1983.
- [Stroustrup,1984] Bjarne Stroustrup: *The C++ Reference Manual*. AT&T Bell Labs Computer Science Technical Report No 108. January 1984. Revised, November 1984.
- [Stroustrup,1984b] Bjarne Stroustrup: *Operator Overloading in C++*. Proc. IFIP WG2.4 Conference on System Implementation Languages: Experience & Assessment. September 1984.
- [Stroustrup,1984c] Bjarne Stroustrup: *Data Abstraction in C*. Bell Labs Technical Journal. Vol 63, No 8. October 1984.
- [Stroustrup,1985] Bjarne Stroustrup: *An Extensible I/O Facility for C++*. Proc. Summer 1985 USENIX Conference. June 1985.
- [Stroustrup,1986] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley, Reading, MA. 1986. ISBN 0-201-12078-X.
- [Stroustrup,1986b] Bjarne Stroustrup: *What is Object-Oriented Programming?* Proc. 14th ASU Conference. August 1986. Revised version in Proc. ECOOP'87, May 1987, Springer Verlag Lecture Notes in Computer Science Vol 276. Revised version in *IEEE Software Magazine*. May 1988.
- [Stroustrup,1986c] Bjarne Stroustrup: *An Overview of C++*. ACM SIGPLAN Notices. October 1986.
- [Stroustrup,1987] Bjarne Stroustrup: *Multiple Inheritance for C++*. Proc. EUUG Spring Conference, May 1987. Also, USENIX Computer Systems, Vol 2 No 4. Fall 1989.

- [Stroustrup,1987b] Bjarne Stroustrup and Jonathan Shupiro: *A Set of C classes for Co-Routine Style Programming*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
- [Stroustrup,1987c] Bjarne Stroustrup: *The Evolution of C++: 1985-1987*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
- [Stroustrup,1987d] Bjarne Stroustrup: *Possible Directions for C++*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
- [Stroustrup,1988] Bjarne Stroustrup: *Type-safe Linkage for C++*. USENIX Computer Systems, Vol 1 No 4. Fall 1988.
- [Stroustrup,1988b] Bjarne Stroustrup: *Parameterized Types for C++*. Proc. USENIX C++ Conference, Denver, CO. October 1988. Also, USENIX Computer Systems, Vol 2 No 1. Winter 1989.
- [Stroustrup,1989] Bjarne Stroustrup: *Standardizing C++*. The C++ Report. Vol 1 No 1. January 1989.
- [Stroustrup,1989b] Bjarne Stroustrup: *The Evolution of C++: 1985-1989*. USENIX Computer Systems, Vol 2 No 3. Summer 1989. Revised version of [Stroustrup,1987c].
- [Stroustrup,1990] Bjarne Stroustrup: *On Language Wars*. Hotline on Object-Oriented Technology. Vol 1, No 3. January 1990.
- [Stroustrup,1990b] Bjarne Stroustrup: *Sixteen Ways to Stack a Cat*. The C++ Report. October 1990.
- [Stroustrup,1991] Bjarne Stroustrup: *The C++ Programming Language (2nd edition)*. Addison-Wesley, Reading, MA. 1991. ISBN 0-201-53992-6.
- [Stroustrup,1992] Bjarne Stroustrup and Dmitri Lenkov: *Run-Time Type Identification for C++*. The C++ Report. March 1992. Revised version: Proc. USENIX C++ Conference. Portland, OR. August 1992.
- [Stroustrup,1992b] Bjarne Stroustrup: *How to Write a C++ Language Extension Proposal*. The C++ Report. May 1992.
- [Stroustrup,1993] Bjarne Stroustrup: *The History of C++: 1979-1991*. Proc. ACM History of Programming Languages Conference (HOPL-2). April 1993. ACM SIGPLAN Notices. March 1993.
- [Taft,1992] S. Tucker Taft: *Ada 9X: A Technical Summary*. CACM. November 1992.
- [Tiemann,1987] Michael Tiemann: "Wrappers:" Solving the RPC problem in GNU C++. Proc. USENIX C++ Conference. Denver, CO. October 1988.
- [Tiemann,1990] Michael Tiemann: *An Exception Handling Implementation for C++*. Proc. USENIX C++ Conference. San Francisco, CA. April 1990.
- [Weinand,1988] Andre Weinand, et al.: *ET++ - An Object-Oriented Application Framework in C++*. Proc. OOPSLA'88. September 1988.
- [Wikström,1987] Ake Wikström: *Functional Programming in Standard ML*. Prentice-Hall, Englewood Cliffs, NJ. 1987. ISBN 0-13-331968-7.
- [Waldo,1991] Jim Waldo: *Controversy: The Case for Multiple Inheritance in C++*. USENIX Computer Systems, vol 4, no 2, Spring 1991.
- [Waldo,1993] Jim Waldo (editor): *The Evolution of C++*. A USENIX Asso-

- ciation book. The MIT Press, Cambridge, MA. 1993. ISBN 0-262-73107-X.
- [Wilkes,1979] M.V. Wilkes and R.M. Needham: *The Cambridge CAP Computer and its Operating System*. North-Holland, New York. 1979. ISBN 0-444-00357-6.
- [Woodward,1974] P.M. Woodward and S.G. Bond: *Algol 68-R Users Guide*. Her Majesty's Stationery Office, London. 1974. ISBN 0-11-771600-6.

索引

索引排序：以数字和符号开头的索引项；以拉丁字母开头的索引项；以中文字开头的索引项（按照汉语拼音顺序排列）。由于原书索引太多（近40页），这里做了一些删节整理。

数字与符号

--, 减量运算符, 189
--, 前缀和后缀, 189
->和委托, 210
->运算符, 185
->*运算符, 190
. 和 ::, 64
. 运算符, 64
. *运算符, 186
. . . . , 省略, 262
//和C兼容性, 63
. :: 显式量化, 327
. :: 和 ., 64
< . . . >语法, 269, 281
<<运算符, 138
=运算符, 138
>>与 template 语法的问题, 282
0, NULL, 176
0, false, 197
0, 空指针, 175
=0语法, 218
1, true, 197

A

Ada语言, XI, 20, 41, 72, 75, 130, 267
Algol 68的声明, 67
Algol 68引用, 56
Algol 68引用重载, 21
Algol 68运算符, 193
ANSI C局部静态数组, 91
ANSI C volatile, 91

ANSI C 标准库, 144
ANSI C 语法, 42
ANSI C++ 标准, 91
ANSI/ISO名字查找, 102
ARM参考手册, 90
ARM名字查找, 100
ARM特征, 93
assert(), 314
Assert(), 314

B

bad_cast, 144
basic_string, 284
bit<N>, 144
bitstring, 144
Booch组件, 284, 301
Booch组件库, 143
bool, 布尔类型, 197
Buffer, 270, 276

C

C enum, 195
C I/O, 66
C++ Release 1.0特征, 44
C++保护模型, 4
C++程序设计环境, 123
C++的较好的C子集, 125
C++的逐步途径, 127
C++和Ada, 149
C++和C, 4, 38
C++ const, 和C, 61

- C++和C的兼容性, 85, 92
 C++和Fortran, 57
 C++和Simula, 73
 C++和Smalltalk, 74
 C++和汇编语言, 85, 151
 C++和类属程序设计, 301
 C++和面向对象的程序设计, 147
 C++和其他语言, 147
 C++和数据抽象, 147
 C++静态类型检查, 147
 C++库, 39
 C++类型检查, 62
 C++连接, 178
 C++流I/O, 137
 C++名字, 38
 C++设计, 35
 C++特征, 91
 C++语法类型系统, 156
 C++异常处理的想法, 303
 C++作为目标语言, 153
 C++作为实现语言, 153
 C++作为通用程序设计语言, 73
 catch, 305
 cerr, 138
 Cfront, 88
 Cfront template, 267
 Cfront template实现, 289
 Cfront, new的实现, 32
 Cfront编译时代价, 41
 Cfront存储使用, 40
 Cfront代码质量, 236
 Cfront的符号表, 40
 Cfront的规模, 40
 Cfront结构, 40
 Cfront虚表的布局, 252
 Cfront资源需求, 39
 char_ref, 58
 char和int, 171
 char和重载, 171
 char类型的常量, 172
 char类型的文字, 172
 cin的初始化, 65
 class, Simula, XI, 21
 class, Simula风格, 2
 class, 保护的单位, 30
 class, 统一的根, 136
 class, template, 269
 class, virtual基, 200
 class, 常规基, 199
 class, 派生, 25
 class, 局部嵌套, 83
 class, 嵌套, 70, 91
 class成员, 225
 class成员顺序依赖性, 100
 class分层结构, 2
 class分层结构的合并, 212
 class分层结构和重载, 174
 class概念, 9
 class根, 202
 class和namespace, 332
 class和struct, 48
 class和template, 282
 class和template, 容器, 266
 class和宏, 容器, 266
 class是用户定义类型, 10
 class作为namespace, 317
 clone(), 228
 CLOS, 207, 210, 233
 Clu, XI, 21, 72, 75, 75, 314
 Clu风格的容器, 267
 CMP, 284
 complex, 51, 144, 291
 complex template, 293
 complex库, 40
 const和Cpp, 336
 const和重载, 171
 const, 为描述界面使用, 60
 const_cast, 262

- A**
 constraints, 273
 const成员初始式, 111
 const成员函数, 221
 const的初始化, 223
 const作为符号常量, 59
 Container模板, 247
 Controlled_container, 283
 copy(), 50
 cout, 139
 cout的初始化, 65
 C变量, 149
 C错误处理, 266
 C代码, 生成的, 236
 C和C++, 69, 130, 151
 C和C++连接, 179
 C兼容性, 85, 92
 C兼容性, 100%, 69
 C兼容性和//, 63
 C兼容性和名字空间, 334
 C兼容性和转换, 173
 C扩充, 106
 C连接, 179
 C声明的语法, 45
 C数值, 117
 C特征, 肮脏的, 9
 C特征, 危险的, 9
 C优点和缺点, 21
 C与C++兼容性, 85, 92
 C语言, 和异常, 304
 C语言的问题, 297
 C语义, 71
 C预处理器, 41
 C预处理器Cpp, 85, 336
- D**
 DAG, 支配, 203
 #define, 337
 delete, free(), 33
 delete和释放, 161
- E**
 delete和析构函数, 161
 delete运算符, 162
 dialog_box, 241
 displayed, 198, 199
 double和float, 170
 dynamic_cast, 241
 dynamic_cast的使用, 231, 253
 dynamic_cast的语法, 242
 dynamic_cast和static_cast, 260
 dynarray, 144
- F**
 false 0, 197
 Fig, 230
 FilePtr, 308
 float和double, 170
 float和重载, 170
 fopen(), 306
 for语句中的声明, 68
 free(), delete, 33
 friend, 错误使用, 235
 friend和成员, 53
 friend和封装, 29
- G**
 GDB_root, 136
 grab_X(), 311
- H**
 Hchar, 119
 histogram, 37

.h文件和template, 299

I

I/O, Ada, 137

I/O, C, 65

I/O, C++流, 137

I/O, printf, C, 138

I/O, 对象, 214

I/O, 对象I/O的实例, 252

I/O, 简洁, 137

I/O, 可扩充, 137

I/O, 扩展字符集, 119

I/O, 类型安全, 137

I/O库, 流, 137

I/O流, 62, 65

if替代#endif, 337

#ifdef, 337

include替代#include, 337

#include, 322, 337

inherited, 226

inline成员函数, 100

inline关键字, 12

inline函数, 11

inline和Cpp, 337

inline和template, 270

intersect Shape, 231

int和char, 170

io, 253

io_counter, 66

io_obj, 253

iocircle, 254

iostream, 201

L

link, 27, 37, 200

list, 37

M

malloc(), new, 32, 61

Map, 253

Matherr, 306

Matrix, 效率, 194

monitor, 32

mutable, 221

N

name(), 249

namespace, 225, 329

namespace, class作为namespace, 317

namespace, template, 284

namespace, using, 321

namespace, 全局, 316, 327

namespace, 无名的, 333

namespace别名, 320, 323

namespace和class, 331

namespace和Cpp, 338

namespace和库, 136

namespace设计目标, 319

namespace语法, 320

narrow(), 264

NDEBUG, 313

network_file_error, 306

new(), 建构函数, 32

new()函数, 10

new, 分配和建构函数, 32

new[]运算符, 162

new_handler, 61

new处理器, 166

new和malloc(), 32, 61

new和分配, 161

new和建构函数, 161

new运算符, 161

No_free_store, 183

noalias, 116

NULL 0, 176

Num, 187

O

Object, 246

- Onfreestore, 183
 OOD, 80, 124
 OOP, 45, 125
 operator delete(), 144
 operator delete(), 继承, 161
 operator new(), 144
 operator new(), 返回值, 166
 operator new(), 继承, 162
 overload声明, 177
- P**
- #pragma, 336
 #pragma, template实例化, 290
 printf, C I/O, 137
 private, 234
 private, public, 29
 private, 继承的实现, 29
 private, 强制到基类, 258
 private运算符, 182
 protected, 235
 protected成员, 235
 Ptr, 186, 286
 public, 236
 public, private, 29
 public, 继承的界面, 30
 public, 10
 pvector, 274
- Q**
- queue, 37
- R**
- raise, 305
 RefNum, 188
 reinterpret_cast, 260
 reinterpret_cast和static_cast, 260
 Release, 1.0, 44
 Release 1.0缺陷, C++, 73
 Release 1.0特征, C++, 44
- Release 1.1, 44
 Release 1.2, 44, 237
 Release 1.3, 88
 Release 2.0, 139
 Release 2.0 =, 52
 Release 2.0和存储管理, 160
 Release 2.0和重载, 170
 Release 2.0事件, 88
 Release 2.0特征, 89
 Release 2.1, 44, 91, 236
 Release 3.0, 44
 Release E, 44
 restrict, 117
 return类型, 229
 return类型, 引用, 57
 return值, operator new(), 166
 return值优化, 234
 ROM, 222
 RTTI, 布局, 252
 RTTI的使用, 245, 246
 RTTI和强制, 242
 RTTI和标准化, 248
 RTTI和库, 136
 RTTI例子, 252
 RTTI设计选择, 254
 RTTI实现复杂性, 241
 RTTI问题, 241
 RTTI误用, 246
- S**
- Season, 196
 set, 217
 set_new_handler, 166
 set_new_handler(), 144
 shape, 45, 216, 252
 Shape*, 286
 Shape*的Set, 286
 Shape, intersect, 231
 signal, 305

- size_t, 162
 slist_set, 217
 sort(), 279
 stack, 9
 static, 贬低全局, 333
 static_cast, 259
 static_cast和const, 259
 static_cast和dynamic_cast, 260
 static_cast和reinterpret_cast, 260
 static_cast和隐式转换, 259
 static成员函数, 224
 static成员和访问控制, 224
 static分配, 禁止, 182
 stdin, 初始化, 65
 stdio.h, 322
 stdout, 初始化, 65
 STL, 144
 string, 144
 String, 53, 54, 56
 string类, 40
 string类, 40
 strtok(), 171
 struct和class, 48
 sum(), 291
- T
- template class, 269
 template complex, 287
 template namespace, 284
 template typedef, 282
 template vector, 269
 template, virtual函数, 269
 template, 编译模型, 298
 template, 参数化类型, 269
 template, 双重发送, 288
 template, 组合技术, 282
 template参数, 非类型, 270
 template参数, 函数, 277
 template参数, 名字空间作为, 280
 template参数, 显式, 277
 template参数, 依赖于, 292
 template参数, 约束, 271
 template参数推断, 276
 template成员, 288
 template的错误检查, 291
 template定义, 查寻, 297
 template定义的上下文, 291
 template关键字, 269
 template函数, 275
 template和.c文件, 297
 template和.h文件, 297
 template和class, 286
 template和inline, 270
 template和virtual函数, 269
 template和编译时间, 289
 template和布局, 270
 template和抽象class, 302
 template和建构函数, 302
 template和库设计, 301
 template和连接时间, 289
 template和内部类型, 303
 template和容器类, 268
 template和唯一定义规则, 291
 template和异常, 302
 template和源代码控制, 289
 template和转换, 287
 template里的名字劫持, 296
 template里的隐式使用, 291
 template例子, 成员模板, 286
 template名字查寻, 293
 template嵌套, 288
 template实例化#pragma, 290
 template实例化, 显式, 290
 template实例化, 自动, 289
 template实例化的上下文, 291
 template实例化指示符, 290
 template实现, 289
 template实现的遮蔽, 290

template实现和继承, 302
 template实现经验, 269
 template实现问题, 269
 template使用经验, 269
 template数组, 269
 template语法, 277
 template语法, >>问题, 281
 template语法, 选择, 281
 template展示台, 297
 template中的歧义性, 293
 template重载, 函数, 278
 template作为template的参数, 271
 terminate(), 144
 this赋值, 61, 161

this和块, 205
 this引用, 17
 this指针, 17
 throw, 304
 true 1, 198
 try, 305
 type_info, 144, 249
 typedef, 24
 typedef, template, 282
 typeid, 248
 typeid的误用, 242
 typeid的使用, 254

unexpected(), 144, 313
 unicode字符集, 118
 union, 262
 Usable, 184
 Usable_lock, 184
 using namespace, 320, 322, 327

V
 vec, 26, 214
 vector, 26, 214

virtual的调用优化, 81
 void*, 161
 void*, 由void*强制, 262
 void*, 转换到void*, 173
 void*赋值, 175
 volatile, ANSI C, 91
 vptr, 48
 vtbl, 48

W

wchar_t, 118
 window, 115, 201
 writeonly, 59
 wstring, 144

汉字开头的索引项

A

安全强制, 242
 安全性和方便性, 326
 安全性和兼容性, 263
 安全性和类型安全性, 62
 安全性和异常, 类型, 304
 安全转换, 173
 按类分配程序, 61, 160
 按成员复制, 182
 按位复制, 182

B

半透明作用域, 29
 包装类, 33
 包装性的设计, 269
 保护, C++保护模型, 4
 保护, class, 单位, 30
 保护界限, 越过, 11
 保护模型, 30
 保守性废料收集, 165
 编码, 低级, 269
 编译, 分别, 2, 33

- 编译程序复杂性, 78
 编译模型, *template*, 297
 编译时间保护, 289
 编译时间和*template*, 289
 编译时间开销, 178
 编译时间类型查询, 195
 编译时间类型检查, 10
 变量, 局部, 11
 变量, 全局, 11
 变量, 未初始化的, 68
 变量, 真正局部的变量, 150
 标志, 结构, 24
 标准, ANSI C++, 91
 标准*virtual*表布局, 97
 标准遍历器, 144
 标准调用序列, 96
 标准废料收集, 168
 标准关联数组, 144
 标准化, ISO C++, 91
 标准化的目标, 127, 132, 96
 标准化和RTTI, 246
 标准库, 322
 标准库, ANSI, 144
 标准库, ISO, 144
 标准库, 缺乏, 88
 标准库中的名字, 248
 标准库中的名字空间, 317
 标准链表, 144
 标准名字编码, 96
 标准容器, 144
 标准是什么, 95
 标准数值库, 144
 标准算法, 144
 标准向量, 144
 标准映射, 144
 标准指令集合, 97
 标准组件库, 137
 表示, 对象的表示, 10, 217
 表示, 遮蔽的表示, 217
 别名, 名字空间, 320, 323
 别名问题, 213
 并行, 2, 7
 并行和异常, 304
 并行支持, 140
 不安全运算符, 168
 不完全类型和强制, 259
 布局, Cfront虚表, 252
 布局, *virtual*基类, 206
 布局, 标准*virtual*表, 96
 布局, 带类的C对象, 16
 布局, 派生类, 28
 布局, 对象, 237
 布局, 多重继承, 203
 布局和*template*, 270
 布局和异常, 313
 布局兼容性, 带类的C, 9
 部分构造的对象, 308
- C
- 参数, *template*作为*template*参数, 271
 参数, 非类型*template*参数, 270
 参数, 关键字, 113
 参数, 过分约束, 272
 参数, 函数*template*, 277
 参数, 名字空间作为*template*参数, 271
 参数, 命名, 113
 参数, 默认, 34
 参数, 显式*template*参数, 278
 参数, 依赖于*template*, 291
 参数的平均数目, 114
 参数的推断, *template*, 276
 参数的值, 235
 参数规则, 放松, 230
 参数化类型, 73, 269
 参数化类型模板, 267
 参数检查, 运行时, 230
 参数名, 113

- 参数匹配规则, 173
参数约束, 对模板, 271
操作, 不安全的, 168
层次结构, *class*, 2
层次结构, 单根的, 124, 137
层次结构, 异常和*class*, 305
插入式废料收集, 165
查询, 编译时类型信息, 280
查找ANSI/ISO名字, 102
查找ARM名字, 99
查找*template*名字, 293
查找名字, 99
常规基*class*, 199
陈列台, *template*, 297
成员, *class*, 225
成员, *protected*, 235
成员, *template*, 288
成员, *template*, *virtual*, 288
成员, *template*例子, 286
成员, 访问控制和*static*成员, 224
成员, 类成员的向前引用, 101
成员, 遮蔽基, 50
成员, 指针指向, 236
成员, 指针指向数据成员, 237
成员初始化顺序, 214
成员初始式, 214
成员初始式, *const*, 111
成员函数, 9
成员函数, *const*, 221
成员函数, *inline*, 100
成员函数, *static*, 224
成员函数, 实现, 16
成员和*friend*, 53
成员顺序依赖性, *class*, 99
程序, 可维护性, 322
程序可靠性, 322
程序设计, C++和类属, 302
程序设计, C++和面向对象, 149
程序设计, 低级, 152
程序设计规则, 低级, 85
程序设计语言, 77, 80
程序组织, 7, 80
持续性符号表, 297
持续性库, 141
重复代码, 273
重复的虚函数表, 234
重排规则, 102
重新编译, 9, 218
重新编译, *virtual*函数, 48
重新编译和异常, 312
重新命名, 212, 319
重载, 31, 170
重载, Algol68引用, 21
重载, 多参数, 173
重载, 反对的理由, 31
重载, 赋值, 34
重载, 函数*template*, 278
重载, 基类和派生, 76, 331
重载, 基于*enum*, 197
重载, 支持的理由, 31
重载和*char*, 171
重载和*class*层次结构, 172
重载和*const*, 171
重载和*float*, 170
重载和*Release 2.0*, 170
重载和继承, 171
重载和连接, 226, 227, 179
重载和名字空间, 328
重载和默认参数, 32
重载和顺序依赖性, 172
重载和效率, 77, 83, 55
重载和转变, 333
重载和转换, 170
重载解析, 170
重载解析, 细粒度, 170
重载匹配, 172
重载运算符, 50
抽象, C++和数据, 148

- 抽象class, 13, 200, 214
 抽象class和template, 302
 抽象class和库, 135
 抽象template和class, 302
 抽象类型, 217
 丑陋的C特征, 7
 初始化, Simula, 32
 初始化, 成员的初始化顺序, 214
 初始化, 动态, 66, 222
 初始化, 静态, 66, 222
 初始化, 库, 66
 初始化, 运行时, 66, 222
 初始化, 运行时的问题, 66
 初始化, 运行时检查的, 256
 初始化, 资源请求, 308
 初始化cin, 66
 初始化const, 222
 初始化cout, 66
 初始化stdin, 66
 初始化stdout, 66
 初始化和分配, 161
 初始化和赋值, 3
 初始化和换页, 66
 初始化和虚拟存储, 66
 初始化顺序, 65
 初始化式, 65
 初始化式, const成员, 111
 初始化式, 成员, 214
 初始化式, 基类, 214
 初始化式, 语法, 302
 处理器, new, 166
 纯virtual函数, 218
 葱头式设计, 254
 存储, 初始化和虚存, 66
 存储, 动态32, 56, 11, 31
 存储, 堆, 11
 存储, 静态32, 56, 11, 31
 存储, 类, 11
 存储, 原始, 114
 存储, 栈, 11
 存储, 只读, 222
 存储, 自动, 11
 存储, 自由11, 56, 11
 存储分配场地, 163
 存储管理, 60, 152, 160
 存储管理, 细粒度控制, 160
 存储管理的调整, 160
 存储耗尽和异常, 309
 存储耗尽控制, 166
 错误, template的错误检查, 291
 错误, 多重定义, 297
 错误, 多重继承, 208
 错误, 可能的, 78
 错误处理, 166
 错误处理, C, 267
 错误处理, 多层, 304
 错误和检查, 301
- D
- 大程序中的命名, 321
 大系统和异常, 309
 代码生成, 236
 代码肿胀的抑制, 273
 代码重复, 273
 带类的C, 5
 带类的C布局兼容性, 6
 带类的C的C兼容性, 16
 带类的C低级特征, 17
 带类的C对象布局, 17
 带类的C静态类型检查, 18
 带类的C没有虚函数, 29
 带类的C实现, 17
 带类的C特征, 17
 带类的C文档, 17
 带类的C预处理器, Cpre, 17
 带类的C运行时支持, 26
 单根层次结构, 125, 136
 单重和多重继承, 198

- 单精度浮点数, 170
派生, 通过派生约束, 272
派生, 禁止, 183
派生, 由int派生, 302
派生, 由内部类型, 302
派生类和基, 48
派生class, 25, 235
派生类, 和重载的基, 330
派生类的布局, 28
派生类重载和基类, 49
派生遮蔽基类, 49
等价, 结构, 13
等价, 名字, 13
等价, 运算符, 187
低级编码, 266
低级程序设计, 153
低级程序设计规则, 85
低级特征, 带类的C, 9
递归下降分析器, 41
调用Fortran, 58
调用规则, 181
调用序列, 标准, 96
动态初始化, 66, 222
动态存储, 11
动态分配和实时, 160
动态和静态类型检查, 73
动态检查异常, 313
动态连接, 155
独立的多重继承, 200
短namespace名字, 323
堆存储, 11
对C++的影响, 操作系统, 3
对template参数的依赖, 291
对象, 部分构造的, 307
对象, 堆栈, 61
对象, 静态, 61
对象I/O, 215
对象I/O, 实例, 252
对象布局, 236
对象布局, 带类的C, 16
对象复制, 228
多参数重载, 174
多层错误处理, 304
多层异常传播, 311
多处理器系统, 140
多态类型, 243
多线程, 140
多重, 库的多重使用, 135
多重方法, 255
多重继承, 75, 198
多重继承, C++, 208
多重继承的布局, 203
多重继承, 独立性, 200
多重继承与废料收集, 208
多重继承的静态检查, 200
多重继承的开销, 208
多重继承的使用, 208, 253
多重继承错误, 203
多重继承的复杂, 203
多重继承工具, 208
多重继承和Simula, 198
多重继承和Smalltalk, 208
多重继承和单继承, 198
多重继承和名字冲突, 198
多重继承和歧义性, 198
多重继承和虚函数, 204
多重继承和异常处理, 306
多重继承实现, 208
多重实例化, 290
- F
- 发送, template双重发送, 288
发送, 双重, 234
翻译单元, 32
翻译限制, 97
返回值优化, 234
访问, 对基类, 235
访问, 通过使用声明调整, 332

- 访问控制, 10, 235
 访问控制, 对建构函数, 63
 访问控制, 对名字, 30
 访问控制和static成员, 224
 访问声明, 30
 访问声明和使用声明, 332
 放松参数规则, 229
 放置, 156, 158
 放置的重要性, 163
 非类型template参数, 269
 非同步事件, 311
 废料收集, 103, 153
 废料收集, 保守式, 165
 废料收集, 标准, 71
 废料收集与多重继承, 208
 废料收集, 反对的理由, 167
 废料收集, 可选的, 147, 167
 废料收集, 支持的理由, 167
 废料收集, 自动, 167
 废料收集和析构函数, 168
 分别编译, 3, 13
 分配, 数组, 162
 分配程序, 按类, 60, 160
 分配的建构函数, new, 32
 分配和new, 161
 分配和初始化, 161
 分配和实时, 动态, 160
 分配场地, 162
 分配, 禁止auto, 182
 分配, 禁止static, 182
 分配, 禁止全局量, 182
 分配, 禁止自由空间, 183
 分析C++, 42
 风格, 混合, 79
 风格, 仅初始化, 55, 67
 风格, 设计风格, 6
 封装和friend, 29
 浮点数到整数的转换, 19
 符号表, Cfront, 40
 符号表, 持续性, 297
 辅助类, 57
 复合, 界面的复合, 323
 复合技术, template, 282
 复合运算符, 194
 复制, virtual, 186
 复制, 按成员, 184
 复制, 按位, 184
 复制, 对象, 228
 复制, 禁止, 182
 复制, 浅, 184
 复制, 深, 184
 复制, 指针, 184
 复制建构函数, 184
 复制控制, 182
 赋值, 184
 赋值, 对this的赋值, 60, 161
 赋值, 对void*的赋值, 177
 赋值的重载, 33
 覆盖, 放松规则, 228
 覆盖和遮蔽, 49

G

- 根class, 200
 根class, 统一的, 136
 更好的匹配, 174
 关键字参数, 113
 关联数组, 253
 关联数组, 标准, 144
 关系运算符, 类型, 255
 规则, 低级程序设计, 85
 规则, 交叉, 174
 规则, 两周, 215
 规则, 零开销, 86
 规则, 名字约束, 291
 规则, 设计支持, 80
 规则, 特殊化, 296
 规则, 唯一定义, 15

- 规则, 一般, 77
 规则, 语言技术, 82
 规则, 语言设计, 76
 规则, 重写, 100
 规则, 重新定义, 76
 规则, 重新考虑, 100
 规则, 重新排序, 100
 过度使用, 继承, 29, 127
 过度使用, 强制, 127
 过分约束的参数, 272
- H
- 函数, `const`成员, 221
 函数, `inline`成员, 100
 函数, `static`成员, 224
 函数, `template`参数, 277
 函数, `template`函数, 275
 函数, `template`和`virtual`, 270
 函数, `template`重载, 278
 函数, `virtual`, 45, 48
 函数, 虚函数表, 48
 函数, 虚函数的实现, 48
 函数, 虚函数的效率, 26, 48
 函数, 不用`virtual`的多态性, 26
 函数, 布局和虚函数, 48
 函数, 成员的实现, 17
 函数, 纯虚, 216
 函数, 调用无定义函数, 19
 函数, 前向, 188
 函数, 强制和指针指向, 260
 函数, 虚函数和多重继承, 203
 函数, 运算符, 54
 函数, 在线, 11
 函数, 只实例化用的函数, 275
 函数, 指针指向, 48
 函数, 转换函数, 54
 函数`new()`, 12
 函数成员, 11
 函数定义, 12
- 函数优化, `virtual`, 183
 函数指针, `virtual`, 48
 函数, `virtual`函数的重新编译, 48
 宏用于类属类型, 27
 环境, C++程序设计环境, 123
 环境, 理想的程序开发环境, 156
 环境, 未来的C++程序设计环境, 123
 唤醒, 论据, 312
 唤醒, 迂回方法, 312
 唤醒和终止, 309
 唤醒语义, 309
 换页和初始化, 66
 回调, 236
 混合模式算术, 51, 170
 混合子, 199
- J
- 基, 初始构造, 220
 基, 初始式, 214
 基, 派生遮蔽, 331
 基, 强制到`private`基, 257
 基, 由`virtual`基强制, 206
 基`class`, 25
 基`class`, `virtual`, 199
 基`class`, 常规, 198
 基`class`, 访问, 235
 基布局, `virtual`, 205
 基成员的遮蔽, 49
 基础库, 140
 基础库, 垂直的, 140
 基础库, 水平的, 140
 基和派生, 47
 基和派生, 重载, 49
 计算, 工程, 130
 计算, 科学, 116, 130, 153
 计算, 数值, 153, 116
 技术, `template`复合, 282
 技术和语言特征, 126
 继承, 多重, 75, 198

- 继承, 过度使用, 29, 124
继承, 通过继承约束, 272
继承operator delete(), 162
继承operator new(), 162
继承private, 实现, 29
继承public, 界面, 29
继承和template, 286
继承和重载, 172
假设, 对异常处理, 304
间接, 186
兼容性, 连接, 85
兼容性和安全性, 266
减量运算符 --, 189
检查, 静态检查的界面, 74
检查, 运行时, 8
检查, 运行时参数检查, 230
检查template的错误, 291
检查和错误, 301
检查异常, 动态, 313
建构, 基类在先, 220
建构函数, new(), 32
建构函数, 对内部类型, 302
建构函数, 访问控制, 63
建构函数, 复制, 184
建构函数, 和new, 161
建构函数, 和new分配, 32
建构函数, 默认, 34
建构函数, 其中的virtual调用, 224
建构函数的调用写法, 265
建构函数和template, 302
建构函数和库, 135
建构函数和异常, 307
建构函数记法, 63
建构函数里的virtual调用, 222
交叉规则, 175
接受的建议, 110
接受的扩充, 110
接受的特征, 110
结构标志, 24
结构等价, 14
解析, 重载, 170
界面, 10
界面, const描述, 59
界面, public继承, 29
界面, template实现和界面, 301
界面, 静态检查, 75
界面, 外国语, 181
界面复合, 323
界面和强制, 135
界面和实现, 217
仅做初始化的风格, 54, 67
禁止auto分配, 183
禁止static分配, 183
禁止派生, 184
禁止复制, 182
禁止全局分配, 182
禁止运算符, 183
禁止自由存储分配, 184
警告, 184
静态初始化, 65
静态存储32, 55, 11
静态对象, 61
静态和动态类型检查, 75
静态检查的界面, 75
静态检查多重继承, 202
静态类型检查, 10, 74, 255
静态类型检查, C++, 148
静态类型检查, 带类的, 18
静态类型检查和库, 135
静态类型检查和设计, 75
静态类型系统, 82
静态异常检查, 312
局部, 默认, 70
局部变量, 11
局部静态数组, 91
局部性, 82
局部性, 嵌套的class, 86

句柄, 188

K

开放的名字空间, 329
 开关, 基于类型, 246
 开关, 基于类型域, 43
 开销, Cfront 编译时间, 41
 开销, 编译时间, 178
 开销, 多重继承, 208
 开销, 连接时间, 178
 开销, 零开销规则, 86
 开销, 运行时间, 178
 科学计算, 116, 130, 152
 可负担的特征, 81
 可见性和访问, 30
 可靠性和异常处理, 311
 可扩充I/O, 137
 可选的废料收集, 147, 167
 可移植性, 88
 空指针0
 控制, 存储管理, 细粒度, 160
 控制, 存储耗尽, 166
 控制, 访问, 10, 234
 控制, 复制控制, 182
 控制, 手工, 86
 库, ANSI标准, 144
 库, Booch组件, 140
 库, C++, 39
 库, complex, 40
 库, ISO标准, 144
 库, 标准库, 322
 库, 标准库名字空间, 316
 库, 标准库中的命名, 219
 库, 标准数值库, 144
 库, 标准组件库, 136
 库, 垂直基础库, 140
 库, 流I/O库, 137
 库, 水平基础库, 140
 库, 语言支持, 135

库, 作业, 7, 136, 140
 库初始化, 65
 库的多重使用, 135
 库和namespace, 135
 库和抽象class, 135
 库和多重继承, 136, 198
 库和建构函数, 135
 库和静态类型检查, 135
 库和类型系统, 135
 库和异常, 135
 库和语言特征, 134
 库和运行时类型信息, 135
 库设计, 134
 库设计和template, 300
 库设计折中, 135
 跨语言连接, 181
 扩充的I/O字符集, 122
 扩充的类型信息, 250
 扩充的字符集, 119
 扩充时的检查表, 108

L

老代码和异常, 312
 老风格强制的记法, 263
 类成员的向前引用, 99
 类的布局, 派生, 28
 类分层结构, 异常, 305
 类型, char常量的类型, 198
 类型, char字面量的类型, 198
 类型, class作为用户定义类型, 9
 类型, return, 228
 类型, template, 参数化, 267
 类型, 参数化类型, 75, 269
 类型, 抽象, 217
 类型, 多态, 243
 类型, 返回引用, 57
 类型, 基于类型的开关, 246
 类型, 内部类型的建构函数, 302
 类型, 内部类型的析构函数, 302

- 类型, 强制和不完全, 259
 类型, 由内部类型派生, 302
 类型bool, 布尔, 197
 类型enum, 196
 类型安全的I/O, 137
 类型安全的连接, 13, 178
 类型安全连接的问题, 181
 类型安全连接的语法, 316
 类型安全性, 62
 类型安全性和异常, 304
 类型编码, 179
 类型标识, 248
 类型查询, 编译时, 279
 类型查询, 运行时, 205, 238
 类型查询运算符, 255
 类型检查, C++, 62
 类型检查, C++静态, 147
 类型检查, 编译时间, 12
 类型检查, 带类的C, 静态, 18
 类型检查, 静态, 10, 73, 254
 类型检查, 静态和动态, 73
 类型检查, 库和静态检查, 135
 类型检查, 默认, 70
 类型检查和设计, 75
 类型检查和设计, 静态, 75
 类型违背, 82
 类型系统, 2
 类型系统, C++语法, 157
 类型系统, 静态, 82
 类型系统和库, 135
 类型信息, 扩充的, 250
 类型信息, 运行时, 205, 238
 类型信息和库, 运行时, 136
 类型域, 基于类型域的开关, 45
 类型域, 显式, 26
 类属程序设计, C++, 300
 类属类型, 宏, 27
 例子, RTTI, 252
 例子, 成员template, 286
 例子, 对象I/O, 252
 例子, 作业, 140
 连接, C, 178
 连接, C和C++, 179
 连接, 动态, 155
 连接, 函数指针, 181
 连接, 跨语言, 181
 连接, 类型安全34, 226, 228, 230, 13, 178
 连接, 增量, 159
 连接程序, 85
 连接程序的问题96, 296, 43
 连接和重载226, 227, 178
 连接兼容性, 85
 连接名, 179
 连接模型, 13
 连接时间和template, 289
 两周规则, 212
 量化, 64
 量化, 显式, 321
 量化::, 显式, 327
 临时量的生存期, 103
 临时量的生存期, 103
 临时量的析构点, 选择, 105
 临时量其他可能的析构点, 105
 临时量删除, 194
 灵活性和效率, 302
 灵活性与简单性, 171
 灵巧引用, 187
 灵巧指针, 186, 286
 零开销规则, 86
 流I/O, 62, 65
 流I/O, C++, 137
 流I/O库, 137

M

- 面向特定系统结构的扩充, 116
 描述, 异常, 313
 名字, C++, 38

- 名字, 全局, 224
 名字编码, 标准, 96
 名字编码和名字空间, 334
 名字查找, 99
 名字查找, ANSI/ISO, 102
 名字查找, ARM, 100
 名字查找, template, 293
 名字冲突, 全局作用域, 316
 名字冲突和多重继承, 211
 名字的访问控制, 30
 名字等价, 14
 名字劫持, 321
 名字劫持, 在template里, 303
 名字空间, 开放性, 329
 名字空间, 嵌套, 329
 名字空间, 转变, 322
 名字空间和C兼容性, 334
 名字空间和版本, 323
 名字空间和名字编码, 333
 名字空间和名字压延, 333
 名字空间和重载, 328
 名字空间污染, 224
 名字空间与标准库, 316
 名字空间作为template参数, 271
 名字连接, 179
 名字前缀, 317
 名字压延, 179
 名字压延和名字空间, 334
 名字约束规则, 291
 名字遮蔽, 49
 命名, 在标准库, 248
 命名, 在大程序里, 321
 命名参数, 113
 模板, Container, 247
 模板, 条件里的模板, 279
 模板的使用, 252
 模板间的关系, 286
 模板设计准则, 267
 模板之间的关系, 286
 模块化, 70
 模块结构, 7
 模型, C++保护模型, 4
 模型, template编译模型, 297
 模型, 保护模型, 29
 模型, 连接模型, 13
 默认, 局部, 69
 默认, 类型检查, 69
 默认, 嵌套, 69
 默认, 私用, 69
 默认参数, 34
 默认参数和重载, 34
 默认建构函数, 34, 303
 默认运算符, 182
 目标, namespace设计目标, 318
 目标, 标准化, 93
 目标, 带类的C, 8
 目标, 异常处理, 304
 目标, 早期库, 136
 目标语言, C++作为, 153
- N
- 内部类型, 和用户定义类型, 11, 82
 内部类型派生, 303
 内部类型的定义, 303
 内部类型的建构函数, 303
 内部类型的析构函数, 303
 内部类型和template, 303
- P
- 排序, 283
 匹配, 更好, 174
 匹配, 重载, 174
 匹配规则, 参数, 174
 评价准则, 对于模板设计, 267
 评价准则, 接受的特征, 36
 评价准则, 扩充, 108
- Q
- 歧义性, 172
 歧义性和多重继承, 199

歧义性, 模板, 292
 异义性控制, 172
 前向函数, 188
 浅复制, 184
 嵌入式系统, 152
 嵌套class, 70, 92, 224
 嵌套class, 局部化, 84
 嵌套class的向前声明, 224
 嵌套template, 288
 嵌套的名字空间, 329
 强制, 安全, 242
 强制, 从virtual基, 206, 245
 强制, 从void*, 262
 强制, 和RTTI, 242
 强制, 新风格, 257
 强制, 隐式, 277
 强制到private基, 258
 强制的过度使用, 125
 强制的问题, 257
 强制的语法, 242
 强制和const, 260
 强制和不完全类型, 259
 强制和界面, 135
 强制和指向函数的指针, 260
 强制记法, 老风格, 262
 强制去掉const, 222, 262
 强制删除, 243
 强制新记法, 257
 取代常规指针, 189
 全局namespace, 316, 327
 全局变量, 11
 全局分配, 182
 全局名字, 224
 全局优化, 81
 全局作用域, 327
 全局作用域名字冲突, 317

R

容器, class和template, 266

容器, class和宏, 266
 容器, Clu风格, 268
 容器, Smalltalk风格, 269
 容器, 标准, 144
 软件部件产业, 140

S

删除临时量, 194
 删除强制, 242
 上下文, template定义的, 291
 上下文, template实例化的, 291
 设计, namespace的设计目标, 318
 设计, namespace的设计思想, 318
 设计, RTTI的替代设计, 254
 设计, template的清理, 268
 设计, template的设计准则, 267
 设计, template和库, 300
 设计, 葱头式, 254
 设计, 库, 135
 设计, 面向对象的, 80, 128
 设计, 语言, 80
 设计规则, 语言, 76
 设计和静态类型检查, 75
 设计和类型检查, 75
 设计者和实现者, 35
 深复制, 184
 声明, Algol, 65
 声明, 实现, 14
 声明, 特殊化, 296
 声明, 在for-语句里, 67
 声明, 在嵌套class里, 前向, 224
 声明, 在条件里, 67
 声明的线性记法, 23
 声明的语法, C, 22
 声明重载, 177
 省略号..., 262
 实例化#pragma, template, 290
 实例化, 多次, 290
 实例化, 手工优化, 291

- 实例化, 推迟, 28, 288
 实例化, 显式template, 290
 实例化, 隐式, 288
 实例化, 自动template实例化, 288
 实例化点, 291
 实例化上下文, template, 292
 实例化指示符, template, 290
 实现, 带类的C, 8
 实现Cfront template, 288
 实现private继承, 29
 实现template遮蔽, 289
 实现virtual函数, 48
 实现成员函数, 17
 实现多重继承, 208
 实现复杂性, RTTI, 238
 实现和界面, 218
 实现和界面, template, 301
 实现问题, template, 267
 实现语言, C++作为, 153
 使用, 通过使用约束, 272
 使用dynamic_cast, 231, 252
 使用RTTI, 245, 246
 使用typeid, 254
 使用的复杂性, 78
 使用多重继承, 208, 254
 使用和误用, 81
 使用模板, 254
 使用声明, 329
 使用声明调整访问, 332
 使用声明和访问指示, 332
 使用声明和使用指示, 325
 使用指示, 320
 使用指示和使用声明, 325
 使用指示和转变, 322
 事件序列, 87
 释放和delete, 161
 释放和清除, 161
 释放数组, 165
 释放问题, 164
 手工实例化优化, 291
 受限指针, 116
 书籍和杂志, 125
 输出运算符, <<, 138
 数据, virtual, 205
 数据成员, 指针指向, 237
 数据抽象, C++, 145
 数组, 标准关联数组, 144
 数组, 关联数组, 254
 数组template, 270
 数组分配, 162
 数组释放, 165
 水平基础库, 140
 顺序, 成员初始化, 216
 顺序, 初始化, 65
 顺序依赖性, 116, 226, 84
 顺序依赖性, class成员, 99
 顺序依赖性, 迂回处理, 66
 顺序依赖性和重载, 172
 私用, 默认, 70
 算术, 混合模式, 52, 170
 碎片, 160
- T
- 特权, 322
 特殊存储器, 160
 特殊地址, 162
 特殊化, 模板, 298
 特殊化规则, 298
 特殊化声明, 296
 特殊化限制, 297
 特征, ARM, 91
 特征, C++ Release 1.0, 44
 特征, C++ Release 2.0, 89
 特征, C++, 93
 特征, 带类的C, 9
 特征, 新的, 207
 提升, 整数, 174
 条件, 在模板中, 279

条件中的声明, 68

通讯, 兄弟类之间, 201

同义词, 319

同义词问题, 212

统一的根class, 136

头文件, 34, 201, 229

头文件, 预编译, 338, 13, 156, 181

推迟实例化, 273, 288

推断template参数, 276

W

外国语接口, 181

危险的C特征, 8

危险的特征, 247

唯一定义规则, 14

唯一定义规则和template, 292

委托, 186

委托, 多重继承, 208

委托, 经验, 209

委托和->, 186

未初始化变量, 67

文件, 头文件, 34, 201, 229, 13, 156

文件, 源文件, 156

文件和template, .c, 297

文件和template, .h, 297

文件和template, 源文件, 297

污染名字空间, 224

无定义函数, 调用, 18

无名的namespace, 332

无约束的方法, 255

误用friend, 234

误用RTTI, 246

误用typeid, 242

X

析构函数, 63

析构函数delete(), 32

析构函数virtual, 164

析构函数, 对于内部类型, 303

析构函数, 显式调用, 165

析构函数和delete, 161

析构函数和废料收集, 168

析构函数和异常, 307

系统, 大系统, 81

系统, 大系统和异常, 309

系统, 混合, 153

系统, 库和类型系统, 136

系统, 嵌入式, 152

系统, 容错系统设计, 304

系统和语言, 15

系统和语言实现, 159

细粒度存储管理的控制, 160

细粒度重载解析, 171

显式template参数, 277

显式template实例化, 290

显式调用析构函数, 165

显式类型域, 27

显式量化, 321

显式量化::, 327

限制, 特殊化, 296

相同, 类型, 248

向量, template, 269

向量, 标准, 144

向前, 类成员的向前引用, 101

向前, 嵌套class的向前声明, 225

效率, 虚函数, 26

效率, 带类的C, 8

效率, 运行时, 11, 133, 254

效率和灵活性, 302

效率和异常, 314

效率和重载77, 83, 54

新风格强制, 257

信号, 312

性能, 160

性能, new, 60

性能, 启动, 66

兄弟class, 201

兄弟类间的通讯, 201

虚, 在建构函数里调用, 219
 虚表布局, 96
 虚成员 template, 288
 虚复制, 184
 虚函数, 45, 48
 虚函数, 纯, 215
 虚函数表, 48
 虚函数和 template, 270
 虚函数和布局, 48
 虚函数实现, 49
 虚函数效率, 25, 49
 虚函数优化, 26, 49
 虚函数指针, 48
 虚函数重新编译, 49
 虚基, 从虚基强制, 206, 245
 虚基 class, 200
 虚基布局, 205
 虚数据, 205
 虚析构函数, 164

Y

一遍编译, 100
 一遍分析, 101
 一致的观点, 107
 一致性维护, 113
 移植问题, 181
 异常, 捕捉的保证, 304
 异常结组, 305, 313
 异常处理, 75, 303
 异常处理, C++的理想, 304
 异常处理, 假设, 304
 异常处理的目标, 304
 异常处理的语法, 305
 异常处理和多重继承, 306
 异常处理和可靠性, 304
 异常的代价, 304
 异常的动态检查, 313
 异常的多层传播, 312
 异常的经验, 311

异常的静态检查, 312
 异常和C, 304
 异常和 template, 301
 异常和并行性, 304
 异常和布局, 313
 异常和抽象的层次, 311
 异常和存储耗尽, 309
 异常和大系统, 310
 异常和建构函数, 307
 异常和库, 311
 异常和老代码, 312
 异常和类层次, 306
 异常和类型安全性, 304
 异常和其他语言, 312
 异常和析构函数, 307
 异常和效率, 313
 异常和溢出, 312
 异常和重新编译, 313
 异常和资源管理, 307
 异常检查, 静态, 313
 异常描述, 313
 溢出和异常, 312
 引用, 56
 引用, const, 56
 引用, return类型, 57
 引用, this, 17
 引用, 灵巧, 187
 引用, 约束, 56
 引用传递, 56
 引用调用, 56
 引用和指针, 56
 引用语义, 148
 隐式, template里的隐式使用, 291
 隐式强制, 276
 隐式实例化, 288
 隐式窄转换, 263
 隐式转换, 51, 170
 隐式转换和 static_cast, 259
 用户定义类型, class是, 9

- 用户定义类型的优先级, 194
 用户定义类型和内部类型, 11, 82
 用户定义运算符, 193
 用户定义运算符的优先级, 194
 优化, 全局, 81
 优化, 手工, 160
 优化, 手工实例化, 291
 优化return值, 236
 优化virtual调用, 81
 优化virtual函数, 183
 优化虚表, 236
 幽雅和简单, 174
 右值和左值, 57
 迂回做法, 对多重方法, 234
 迂回做法, 对关键词参数, 115
 迂回做法, 对唤醒, 311
 迂回做法, 对顺序依赖性, 66
 与实现有关的行为, 95
 语法, 84
 语法, < . . . >, 269, 281
 语法, =0, 218
 语法, C声明, 22
 语法, dynamic_cast, 245
 语法, namespace, 320
 语法, template, 277
 语法, template的>>问题, 281
 语法, template的选择, 281
 语法, 初始式, 302
 语法, 类型安全的连接, 316
 语法, 类型系统, C++, 156
 语法, 强制, 242
 语法, 异常处理, 304
 语法, 指针, 237
 语法的重要性, 25
 语法中的冗余, 305
 语言, C++作为目标语言, 154
 语言, C++作为实现语言, 154
 语言和环境, 分离, 148
 语言和实现, 90
 语言设计规则, 76
 语言特征和技术, 126
 语言特征和库, 134
 语义, C, 71
 语义, 唤醒, 309
 语义, 引用, 148
 语义, 值, 148
 语义, 指针, 148
 语义, 终止, 309
 原始存储, 161
 源程序, C++, 156
 源程序, 控制和template, 289
 源程序文件, 156
 源程序文件和template, 297
 源程序正文, 合法, 95
 源代码里的名字, 319
 约束, 对template参数, 271
 约束, 通过派生, 272
 约束, 通过继承, 272
 约束, 通过使用, 273
 约束, 引用, 57
 约束规则, 名字, 291
 约束和错误检查, 272
 约束和可读性, 272
 运算符--, 减量, 189
 运算符->, 186
 运算符->*, 191
 运算符**, 指数, 191
 运算符*^, 指数, 193
 运算符, 191
 运算符, private, 182
 运算符, 禁止, 182
 运算符, 类型关系, 255
 运算符, 类型获取, 255
 运算符, 用户定义, 193
 运算符, 用户定义优先级, 194
 运算符, 187
 运算符*, 191
 运算符++, 增量, 189

- 运算符<<, 138
 运算符<<, 输出, 138
 运算符=, 184
 运算符delete, 10, 160, 164
 运算符delete[], 162
 运算符new, 11, 160
 运算符new[], 162
 运算符等价, 186
 运算符复合, 194
 运算符函数, 54
 运算符默认, 182
 运算符重载, 50
 运行时, 检查的初始化, 256
 运行时保证, 31
 运行时参数检查, 234
 运行时初始化, 65, 222
 运行时初始化, 问题, 65
 运行时检查29, 42, 19
 运行时开销, 178
 运行时类型获取, 47
 运行时类型信息, 205, 238
 运行时类型信息和库, 135
 运行时效率, 11, 133, 255
 运行时支持, 带类的C, 25
 运行时支持, 最小化, 3
- Z
- 增量连接, 155
 增量运算符++, 189
 窄转换, 19, 173
 窄转换, 隐式, 263
 找到template定义, 297
 遮蔽, 和重载, 49
 遮蔽template实现, 289
 遮蔽表示, 217
 遮蔽基, 派生类, 331
 遮蔽基成员, 49
 遮蔽名字, 49
 真正的局部变量, 149
- 整数提升, 174
 正交性, 79
 支持, 对库的支持, 75
 支持, 设计支持规则, 80
 支持, 语言对库的支持, 135
 支持并行性, 140
 支持库, 238
 值传递, 56
 值调用, 56
 值语义, 149
 只读存储器, 222
 指示符, template实例化, 290
 指数运算符**, 191
 指数运算符*^, 193
 指针, 到成员, 237
 指针, 到函数, 48
 指针, 到函数的指针和连接, 181
 指针, 到函数的指针和强制, 260
 指针, 到数据成员, 237
 指针, 灵巧, 186, 286
 指针, 取代常规, 189
 指针, 受限的, 116
 指针0, 空, 175
 指针this, 17
 指针复制, 184
 指针和非指针, 10
 指针和引用, 57
 指针语法, 237
 指针语义, 149
 中断, 312
 终止的论据, 310
 终止和唤醒, 309
 终止语义, 309
 重定义规则, 100
 主要和次要扩充, 238
 转变, 78, 214, 264
 转变到名字空间, 322
 转变到新强制, 265
 转变和使用指示, 322

- 转变和重载, 328
转变路径, 113
转换, 65
转换, static_cast和隐式, 259
转换, 安全, 173
转换, 标准, 174
转换, 浮点数到整数, 19
转换, 隐式, 52, 170
转换, 隐式窄转换, 264
转换, 窄转换, 19, 173
转换到void*, 172
转换格, 174
转换函数, 54
转换和C兼容性, 173
转换和template, 287
转换和重载, 170
转换图, 174
资源管理, 162
资源管理和异常, 307
资源获取, 初始化, 308
资源需求, Cfront, 39
字符集, 117
字符集, 8位, 121
字符集, I/O扩充, 121
字符集, unicode, 121
字符集, 国家, 117
字符集, 扩充, 118
自动template实例化, 288
自动存储, 11
自动废料收集, 167
自动原型, 18
自赋值, 33
自由存储32, 55, 11, 32
自由存储分配, 禁止, 182
组合方法, 207
左值与右值, 57
作业的例子, 139
作业库, 7, 136, 139
作用域, 半透明, 30
作用域, 名字冲突, 全局, 317
作用域, 全局, 327
作用域和Cpp, 337